

版权注意事项：1、书籍版权归著者和出版社所有；  
2、本PDF仅用于个人获取知识，进行私底下知识交流；  
3、PDF获得者不得在互联网以任何目的进行传播；  
如有需要，请尽量购买正版实体书！支持书籍作者！！





# 高性能

服务系统 / 构建与实战  
高性能 / 高可用 / 可伸缩

## 服务系统构建与实战

银文杰 / 编著

## 银文杰



笔名：说好不能打脸

博客地址：[blog.csdn.net/yinwenjie](http://blog.csdn.net/yinwenjie)

资深IT码农，最大爱好就是敲敲代码，写写博客，研究研究创业热点。CSDN博客作者、CSDN Java EE知识库特约编辑。曾参与电信行业、物流行业多个核心系统建设，对系统顶层设计、技术线路规划、业务系统性能调整有较丰富的经验；也曾有几年头脑发热拍案创业，兼职市场销售、电话客服、公司保安以及清洁大叔。

# 高性能 服务系统构建与实战

银文杰 / 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING



## 内 容 简 介

影响业务系统性能的因素很多,计算机系统的各个层面都有涉及:从硬件、网络、操作系统、中间件、存储,直到自身代码质量。所有技术团队都曾为解决性能问题、提高性能峰值绞尽脑汁,从千头万绪到生不如死。本书基于作者 10 余年工作经历中踩过技术神坑,总结整理而成。虽然不能将计算机系统各个层面中影响性能的因素全部介绍完,但还是希望通过讨论业务系统负载层、网络通信层解决性能问题的过程,启发读者,为读者在工作中解决性能问题提供借鉴思路。

本书适合计算机软件领域中立志在架构师职业路线上长期发展的技术人员阅读,无论读者是有一定工作经验的软件工程师、运维工程师还是在校大学生,都适合阅读本书。本书知识点横跨系统架构领域和软件架构领域,所以为了更好地阅读本书,读者最好曾经使用过 Linux 操作系统,也最好有 Java 编程语言的使用能力。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

## 图书在版编目(CIP)数据

高性能服务系统构建与实战 / 银文杰编著. —北京:电子工业出版社, 2017.7

ISBN 978-7-121-31509-1

I. ①高… II. ①银… III. ①软件设计 IV. ①TP311.1

中国版本图书馆 CIP 数据核字(2017)第 105157 号

策划编辑:付 睿

责任编辑:石 倩

印 刷:三河市鑫金马印装有限公司

装 订:三河市鑫金马印装有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本:787×980 1/16 印张:27.5 字数:595 千字

版 次:2017 年 7 月第 1 版

印 次:2017 年 7 月第 1 次印刷

定 价:89.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式:010-51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 序 言

科学的学习方法将大脑的状态分为三个区：舒适区、学习区和恐慌区。在舒适区中你可以基于自己熟悉的知识去做一些习以为常的事情，因为你已经具备了处理这些事情所需要的知识，所以做这些事情一般都会很得心应手。大脑状态处于恐慌区时给你的体验和舒适区刚好相反，由于你不具备处理这些事情的任何知识，或者说处理这些事情已经超出了你最大的知识范围，所以你会对事情的结果感到不确定，甚至沮丧、焦虑、崩溃、放弃。处于恐慌区时是不利于学习的，因为你的大脑思维不能被顺序整理，不能被归纳总结。

学习区又叫作脱离舒适区，处于这个区间的大脑，在做对应的事情时会感觉到挑战，并处于亢奋状态。让大脑进入学习区的事情都有这样的特点：你的大脑可以利用既有知识引申总结新的知识，并对自身知识树的缺失部分进行补全。所以，**让大脑脱离舒适区进入非舒适区是个人能力进步的一个根本要素**。例如，你可以使用 Java 进行编程活动，熟练自如后再在这个基础上学习 Groovy、Scala 等编程语言；再例如，你拥有了自己的编程习惯，再在此基础上融入别人的编程方法。以上两个例子都是在同一知识领域体系下的大脑状态区域平移。你也可以让你的大脑状态在不同的知识领域下进行平移，例如做开发的朋友可以去尝试做产品团队、市场团队的一些工作。我的偶像，逻辑思维的罗振宇老师对此有一个非常棒的总结：持续地做你不擅长的事。

总之，不能让自己的大脑在舒适区待得太久。在舒适区待久了的人也有一些共同表现，例如对新生事物天生持抵制态度，听不进去别人的建议，在职场“混资历”，甚至看不得别人取得任何成绩。总之如果有一天你发现自己听到类似“我都工作 20 年了，什么没见过？”“像你这样的项目，我当年一个人带 20 个！”这样的话，那么讲这种话的人一定是一个让自己在舒适区待得太久的人。不要混资历，不要用你的战术勤奋掩盖你的战略懒惰。

银文杰

2017 年 5 月

# 前言

本书主要的代码示例采用 Java 写成，对于一些相对独立章节中的代码，笔者将其整理后形成示例工程。例如实战章节中的日志采集工程、图片服务工程，笔者已经上传到了 CSDN 的线上资源管理中，可供读者自行下载。本书一共分为四个部分，第一部分对日常开发任务中经常遇到的问题进行了总结，并将这些问题分类，分解出这些问题在整个软件架构中的位置。第二部分、第三部分和读者一起讨论软件架构中的负载层性能设计、业务层性能设计并穿插讲解了一些存储层的设计关注点，其中将详细讨论一些具体的软件/组件应用以及它们的工作原理。第四部分为实践章节，这一部分将基于已经介绍过的知识点和读者一起将它们用于工程实战，对于之前没有涉及的新知识点，也会在其中进行简要说明。

本书大量使用操作系统、Java 知识体系、软件设计中的基础知识，包括但不限于：操作系统线程原理、悲观锁/乐观锁、软件设计模式等。例如本书中至少使用的设计模式包括：命令模式、构建者模式、观察者模式、责任链模式；本书中至少涉及的 Java 基础知识包括：有限/无限队列、悲观锁/乐观锁、SPI 规则、concurrent 工具包、状态机；本书还关联至少如下第三方组件：分布式文件系统、Redis、关系型数据库、Keepalived、ZooKeeper。因为篇幅所限，本书并不可能用太多的文字对这些基础知识、第三方组件进行详细介绍，甚至不会专门说明某些技术点。所以本书更适合有一定一线业务系统开发经验的软件工程师阅读，并且在工作过程中使用过 Linux 系列操作系统（最好是 CentOS），因为本书讲解的知识点、介绍的安装运行方法、讨论的工作原理、描述的操作过程、给出的示例代码环境全部都是基于 Linux 操作系统的。如果你想从一名开发人员成长为一名软件架构师，那么本书绝对是你合适的一块“垫脚石”；本书还适合有一定系统运维工作经验的 IT 工程师阅读，如果你想完成从传统的 IOE 系统运维到移动互联网系统领域运维的蜕变，那么本书所介绍的知识也会给你一定的启发。

由于本书内容较丰富，文字讲解部分就占用了相当的篇幅，所以为了尽可能节约篇幅，本书在列举代码段落时往往只保留了主要的代码片段，并以“……”表示代码段落中有省略的片

段。另外，如果在代码片段中使用 Java 标准注释规范，则将占据多余的空间，所以本书大部分使用了 Java 中的单行注释方式对代码目的进行说明。类似成体系的工程示例，如日志采集案例、图片处理案例等，都在相关章节中注明了完整的工程示例下载地址，以便读者查看更详尽的实现方法。最后，本书中多数图片由笔者自行绘制（90%以上），有一部分图片来源于互联网资源，凡是后者笔者都在图片下方进行了明确的说明。

本书成书于笔者对自己博客文章的整理，其中有 30% 的内容为成书整理时新增。在这个过程中有很多朋友给予笔者帮助，帮助笔者校正博客文章中的错误。特此感谢以下网友（CSDN 账号，排名不分先后）：amadis\_chen、自然的发呆、thisisgpy、github\_33423142、shadabing、fyc198610、zkq1989、sinat\_25444367、Tony\_tec、周创、gongfengying、z3133464733、qq\_32159081、weixin\_33750642、fengyong7723131、白糖、a35946729、zzpapzzp、qq\_16387501、LX\_871225、littlebugu。

笔者还要感谢家人，他们都以自己的方式在笔者写作期间默默地给予支持。笔者最后还要特别感谢电子工业出版社博文视点编辑付睿老师和参与本书校对整理工作的各位编辑，没有他们的勤劳付出就不会有本书的出版发行。

---

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31509>



# 目 录

## 第一部分 前序

第 1 章 那些年一起踩的坑.....	2
1.1 性能问题.....	2
1.2 可用性问题.....	3
1.3 异常处理问题.....	4
1.4 系统间依赖问题.....	4
1.5 系统雪崩问题.....	7
第 2 章 业务系统分解.....	9
2.1 负载层技术.....	10
2.2 业务层技术.....	12
2.3 存储层技术.....	13

## 第二部分 负载层技术与设计

第 3 章 Nginx 技术.....	16
3.1 Nginx 中的基本技术理论.....	16
3.1.1 一致性 Hash 算法.....	16
3.1.2 轮询与加权轮询.....	18
3.2 Nginx 的安装和使用.....	20
3.3 Nginx 的重要配置讲解.....	22



3.4 Nginx 的重要设置.....	25
3.4.1 use [ kqueue   rtsig   epoll   select   poll ].....	25
3.4.2 worker_processes 和 worker_connections.....	26
3.4.3 max client 的计算方式.....	29
3.5 Nginx 的常用模块.....	30
3.5.1 gzip 压缩模块.....	30
3.5.2 rewrite 模块.....	32
3.5.3 健康检查模块.....	34
3.5.4 图片动态缩略模块.....	37
第 4 章 LVS 技术.....	41
4.1 网络协议基础知识.....	41
4.1.1 链路层报文.....	42
4.1.2 网络层 IP 报文.....	42
4.1.3 传输层 TCP 报文.....	44
4.2 LVS 的三种工作方式.....	45
4.2.1 LVS-NAT 工作方式.....	45
4.2.2 LVS-DR 工作方式.....	47
4.2.3 LVS-TUN 工作方式.....	49
4.2.4 LVS 调度方式.....	52
4.3 LVS 设置实战.....	53
4.3.1 LVS-NAT 方式设置.....	53
4.3.2 LVS-DR 模式设置.....	57
4.3.3 ipvsadm 参数汇总.....	60
第 5 章 其他负载层技术.....	63
5.1 DNS 和智能 DNS.....	63
5.2 CDN 网络.....	65
5.3 Keepalived.....	67
5.4 不得不提的 Tengine.....	68

第 6 章 负载层性能实战 .....	69
6.1 负载层技术实战场景 .....	69
6.1.1 负载场景一 .....	69
6.1.2 负载场景二 .....	70
6.1.3 负载场景三 .....	71
6.1.4 负载场景四 .....	72
6.2 方案一：使用 Nginx 初步解决性能瓶颈问题 .....	72
6.3 方案二：使用 LVS + Keepalived + Nginx 增加吞吐量和稳定性 .....	74
6.4 方案三：使用 DNS 和 CDN 网络优化整体性能 .....	75

### 第三部分 系统间通信

第 7 章 系统间通信：网络 I/O 模型 .....	78
7.1 模型 .....	78
7.1.1 信息格式 .....	79
7.1.2 网络协议 .....	80
7.1.3 通信方式/框架 .....	82
7.2 网络 I/O 模型：阻塞模式 .....	82
7.2.1 通信模型概要 .....	82
7.2.2 阻塞模式深入分析 .....	87
7.2.3 问题的根源 .....	91
7.3 网络 I/O 模型：同步非阻塞模式——对阻塞模式的改进 .....	93
7.3.1 首次改进 .....	97
7.3.2 再次改进 .....	99
7.3.3 依然存在问题 .....	101
7.4 网络 I/O 模型：多路复用（I/O Multiplex） .....	101
7.4.1 典型的多路复用 I/O 实现 .....	102
7.4.2 Java 对多路复用 I/O 技术的支持 .....	103
7.4.3 Java NIO 框架简要设计分析 .....	112
7.4.4 Java 实例改进 .....	114
7.4.5 多路复用 I/O 的优缺点 .....	118
7.5 网络 I/O 模型：异步 I/O .....	119

7.5.1 Java 对 AIO 的支持.....	120
7.5.2 Java 提供的 AIO 支持示例 .....	122
7.5.3 还有改进可能 .....	128
7.6 第三方组件: Netty.....	128
7.6.1 为什么需要 Netty .....	129
7.6.2 Netty 快速上手.....	130
7.6.3 Netty 中的重要概念.....	135
7.7 再次审视 Netty 的作用.....	141
7.7.1 对网络 I/O 模型的封装 .....	142
7.7.2 对数据信息格式的封装 .....	143
7.7.3 解决了“技术层”框架中的技术问题.....	146
7.7.4 解决半包问题和粘包问题 .....	148
7.8 不得不提的线程池 .....	152
7.8.1 为什么要使用线程池 .....	152
7.8.2 线程池基本使用 .....	155
7.8.3 ThreadPoolExecutor 逻辑结构和工作方式.....	156
7.8.4 线程池的等待队列 .....	159
7.8.5 拒绝任务 .....	165
7.8.6 ThreadPoolExecutor 中常用属性总结.....	168
第 8 章 RPC 与系统间调用 .....	170
8.1 RPC 技术原理 .....	170
8.1.1 什么是 RPC.....	170
8.1.2 RPC 要素.....	171
8.1.3 更泛化的 RPC 定义.....	173
8.1.4 典型的 RPC 框架介绍.....	174
8.1.5 RPC 框架的性能依据 .....	175
8.2 RPC 实践: Apache Thrift 基本使用 .....	176
8.2.1 IDL 格式概要.....	177
8.2.2 简单的 Apache Thrift 代码 .....	181
8.3 RPC 实践: Apache Thrift 深入分析 .....	185
8.3.1 Apache Thrift 与消息格式 .....	185

8.3.2 Apache Thrift 与通信模型 .....	190
8.3.3 Apache Thrift 与线程池 .....	193
8.4 RPC 实践：解决异常问题 .....	193
8.4.1 分布式业务的异常解决思路 .....	195
8.4.2 事务补偿的简单实现 .....	201
8.5 SOA 和服务治理 .....	224
8.5.1 SOA 概述 .....	225
8.5.2 ESB 概述 .....	227
8.5.3 常见的 ESB 产品 .....	229
8.5.4 服务治理框架 .....	231
<b>第 9 章 系统间通信：消息队列技术 .....</b>	<b>237</b>
9.1 消息队列原理 .....	237
9.1.1 消息 .....	237
9.1.2 服务结构 .....	238
9.2 消息协议 .....	238
9.2.1 XMPP 协议 .....	239
9.2.2 Stomp 协议 .....	241
9.2.3 MQTT 协议 .....	244
9.2.4 AMQP 协议 .....	248
9.2.5 不得不提的 JMS 规范 .....	251
9.3 MQ 实践：ActiveMQ 基本概念和使用 .....	253
9.3.1 ActiveMQ 的简易安装过程 .....	253
9.3.2 ActiveMQ 的其他命令参数 .....	255
9.3.3 在 ActiveMQ 中传递 Stomp 消息 .....	256
9.3.4 ActiveMQ 中的 Queue 和 Topics .....	258
9.3.5 JMS 和协议间转换 .....	260
9.3.6 持久化消息和非持久化消息 .....	266
9.3.7 持续订阅和非持续订阅 .....	267
9.4 MQ 实践：ActiveMQ 性能优化 .....	267
9.4.1 ActiveMQ 性能优化思路 .....	267
9.4.2 ActiveMQ 中的网络配置 .....	268

9.4.3 ActiveMQ 处理规则和优化.....	273
9.4.4 ActiveMQ 的持久消息存储方案.....	285
9.5 MQ 实践: ActiveMQ 集群方案 .....	299
9.5.1 ActiveMQ 高性能方案 .....	300
9.5.2 ActiveMQ 高可用方案 .....	311
9.6 其他 MQ 技术: Apache Kafka .....	321
9.6.1 Kafka 设计概要.....	321
9.6.2 Kafka 集群安装: 配置过程.....	333
9.6.3 Kafka 常用命令.....	336

## 第四部分 场景实战

第 10 章 场景实战: 其他储备知识 .....	340
10.1 数据存储 .....	340
10.1.1 块存储 .....	341
10.1.2 共享存储/共享文件存储.....	343
10.1.3 对象存储系统 .....	344
10.2 磁盘阵列系统 .....	345
10.2.1 RAID 0.....	346
10.2.2 RAID 1.....	347
10.2.3 RAID 10 和 RAID 01 .....	348
10.2.4 RAID 5.....	349
10.3 NoSQL 技术 .....	351
第 11 章 场景实战: Kafka 与日志采集.....	355
11.1 Kafka 应用场景: 场景说明.....	355
11.2 Kafka 应用场景一: 侵入式方案 .....	357
11.2.1 设计重点 .....	358
11.2.2 编码过程: 生产者和业务系统集成.....	361
11.2.3 是否使用 Spring Integration-Kafka.....	366
11.2.4 编码过程: 消费者端.....	367
11.3 Kafka 应用场景二: 调整侵入式方案 .....	371

11.3.1	方案一的问题所在	371
11.3.2	方案二的解决思路	371
11.3.3	方案二的主要代码示例	377
11.3.4	其他设计思考	380
11.3.5	百度站长统计工具	382
11.4	Kafka 应用场景三：非侵入式方案	383
11.4.1	Apache Flume 介绍	383
11.4.2	设计方案	384
11.4.3	配置过程概要	386
11.4.4	方案三细节说明	388
第 12 章 场景实战：图片服务		392
12.1	需求场景	392
12.2	概要设计阶段	393
12.2.1	分布式文件系统选型	394
12.2.2	缓存系统选型	395
12.2.3	路由层选型	397
12.2.4	架构设计细化	400
12.2.5	其他技术选型	401
12.3	关键技术点考量	403
12.3.1	责任链模式	403
12.3.2	Redis 中的数据结构选择	404
12.3.3	使用 Spring Boot	406
12.3.4	其他技术特性	408
12.4	详细设计阶段	412
12.4.1	位图基本知识	412
12.4.2	Nginx 中的 Proxy Cache 配置	418
12.4.3	责任链进行图片处理	420
12.4.4	Redis 缓存操作	423

# 第一部分

## 前序

本部分将介绍在实际工作中技术人员最容易遇到的和系统可用性、系统高性能有关的一些问题，使读者对本书将要讨论的问题有一个概要性的理解，这样便于各位读者选择性地阅读本书章节。第 1 章中将分类讨论这些问题的由来，描述的这些问题场景也将越来越会对软件系统造成更大的伤害。第 2 章中将对绝大多数业务系统进行具有共性的层次分解，并结合第 1 章的内容介绍每一层最可能出现的问题。

# 第 1 章

## 那些年一起踩的坑

翻开此书的读者一定是一名软件技术人员。无论职位高低、技术生熟、资历深浅，一名软件技术人员一定是从一个一个软件项目/产品中熬出来的：各种系统磨难造就了今天的你。可能你夜以继日写代码，只为了能按时完成 JIRA（Atlassian 公司出品的项目与事务跟踪工具）每一次发布；可能你是一名公司的救火队员，坐着红眼航班从一个火场奔向另一个火场；可能你是带领着一帮项目/产品团队的技术人员，一上班就坐在会议室和负责需求的同事讨论，然后在对方下班时感叹：今天又要加班写代码了；你还可能被上级要求电话 24 小时开机，而且确实常常在熟睡时被叫醒——因为接口又调不通了；这都算好的，更悲催的是由于莫名其妙的脏数据问题，所以团队被要求每周都要手工进行一次数据比对和清理，无论是盘点订单数据、交易数据、库存数据还是用户积分；或者你比较幸运，被公司选中设计新系统，但是却无意发现下面的程序员经常抱怨接口故障率太高，经常为此加班……

无论你在公司扮演什么样的技术角色，以上类似的问题肯定是遭遇过的。实际上大多数情况下单个系统独立工作都是没有问题的，或者说不会出现很多难以解决的问题。而一旦一个业务需要多个系统联合工作才能完成，那么往往就会出现各种问题了。

### 1.1 性能问题

读者可能遇到过这样的问题，线上的某系统某功能模块的 TPS/OPS（每秒请求/事务数量）时常保持在几千上下，监控环境和系统日志所表现出的结果都是正常的。但是随着业务的成熟，这个系统模块的 TPS 开始出现上万的情况。这时系统就会出现莫名其妙的错误，要么就是大量客户端请求超时，要么就是磁盘 I/O 尤其是写操作时特别高，要么就是数据库出现无数的表锁。引起这些问题的原因却多种多样，有可能是 Linux 操作系统的若干参数没有优化，有可能



是基础设施薄弱，还有可能是数据表设计阶段没有联系业务方进行聚集索引和非聚集索引的调整。这些问题的最终结果就是导致系统无法承受较猛烈的流量洪峰。

性能问题的调优涉及多个方面：硬件层面、网络层面、操作系统层面、应用软件层面和代码质量层面。例如，根据笔者经验，如果使用 Sonar 进行代码审查出现圈复杂度较高的代码片段，那么这些代码片段不仅难以维护，而且代码性能一般也不会太好。如果使用 JVM 系列语言（例如 Groovy）来编写高性能的代码，那么首先就应该熟练线程/线程池的定义和使用方式。因为类似 JVM 系列的语言都直接采用了操作系统层面的线程和锁的概念来管理并发性，并在业务集中点优先选用乐观锁解决冲突。再例如，如果你使用了 JVM 承载业务服务，但是 JVM 的垃圾回收器没有做调整，例如没有使用-server 模式，那么这个业务服务的性能肯定不会太好。再举一个基础设施层面的例子，首先技术团队必须注重基础设施建设，使用磁盘阵列 RAID1+0 方案为数据库搭建的块存储设施，在实际应用方面肯定强于单块企业级硬盘，当然如果是使用类似阿里的云存储方案，就不需要担心这个问题，因为别人已经帮你做了。

以上提到的这些“质量门”，本书当然不可能全部进行讲解（但是笔者的个人博客对这些问题都有分析，可参见 <http://blog.csdn.net/yinwenjie>）。本书将依据如下的思路来讲解性能问题的优化方式：既然无法减轻整个系统的性能压力，那么就在系统内部分散压力并且缓存待完成的任务。分散压力到多个服务节点，然后从系统原理的各个层面优化每一个服务节点处理单个请求的性能，最终达到提升整个系统性能的目的，不能立即达到这个目的也没关系，至少要保持服务节点持续工作，按照自身资源的限制能力对任务进行异步处理。这就是本书的两部分内容：负载均衡技术和消息队列技术所要达到的讲解效果。

## 1.2 可用性问题

技术人员经常会遇到这样的问题，一个已经上线的业务系统突然宕机。但是又没有可接替它继续工作的备用系统。这种情况就算单个服务节点的性能再高也没有任何意义。高可用性是系统建设的另一个方面，不是说让性能无限制地“高下去”就可以了，而且还要注意系统的容灾容错性。试想一下一个可以承受较高吞吐量峰值的业务系统，用着用着突然宕机且不能在短时间恢复了怎么办。对于客户来说这样的“高性能”是没有意义的，因为它最终也是不可用的，所以高可用性的前提是：保证服务系统能够持续工作。

实现高可用性一般有两种手段：一种是通过第三方软件/组件保证系统的可用性；另一种是软件/组件自身已具备高可用的技术实现。例如我们经常使用 Keepalived 来监控和热切换两台 LVS 节点的工作状态，保证当某台 LVS 节点宕机后（虽然实际经验中它不怎么宕机），另一台能立刻接替它的工作。而有的分布式系统自带高可用解决方案，例如类似 HDFS 的分布

式文件系统一般依靠副本技术实现高可用性，当一个副本不可用时就会有另一个副本接替工作。ZooKeeper 依靠选举策略保证分布式协调器的高可用性，当一个 Leader 节点不可用时，剩下的 Follower 节点会触发选举动作，重新选举一个新的 Leader 节点来领导分布式系统工作。

由于本书中应用了很多第三方组件，所以本书中也会介绍这些第三方组件的高可用集群搭建方案，为读者解决单节点业务系统中不可避免的高可用性问题提供帮助。例如本书会讲解负载均衡层的多种高可用方案、消息队列系统的高可用方案，以及在实战案例中如何运用这些方案。

### 1.3 异常处理问题

这里的异常处理当然不是指我们在写代码时，对代码运行时可能抛出的异常进行解决的过程（经验告诉我们，代码级别对异常捕获处理不充分，甚至出现异常吞噬的情况，往往会加剧这里所讨论问题的严重性），这类异常处理问题主要是指同一顶层设计下，多个子系统间用来调用时发生的异常问题。在分布式业务执行过程中，技术人员需要假设的前提是：调用过程的异常问题无论如何都会出现，即使是再完美的业务系统也不能避免。

系统间调用的异常如果处理不好，那么其造成的损害，特别是对数据一致性的损害将是巨大的。例如财务系统的实收账款已经增加，但是给纳税系统的开票申请却迟迟没有发出去。那么如何保证这些异常被正确处理就是技术人员需要考虑的，本书提供了多种思路来处理这些异常问题，例如使用消息队列的事务机制、重发机制或者私信机制，利用 RPC 实现的事务补偿机制以及使用带有熔断机制的第三方组件。另外说明一点，虽然本书没有介绍记录日志的具体手段，但是日志数据确实是排查和解决系统间调用异常的重要手段，或者说是非常重要的手段。

### 1.4 系统间依赖问题

系统间依赖问题是我们遇到的另一类严重问题，正常情况下整个顶层设计下的若干个子系统肯定存在依赖关系，但是依赖关系是否科学、是否可以持久维护是需要考虑的一个非常关键的要点。如图 1-1 所示的系统依赖结构就是一个好的依赖结构。

图 1-1 中的两种依赖关系都是正确的示例：A 元素正常工作依赖于 B 元素的正常工作，或者 A 元素的正常工作依赖于 B、C、D 元素的正常工作。这里的 A、B、C、D 四个元素可以指代四段代码，也可以指代一个业务系统中的四个功能模块，还可以指代顶层架构设计中的四个独立工作的业务系统。

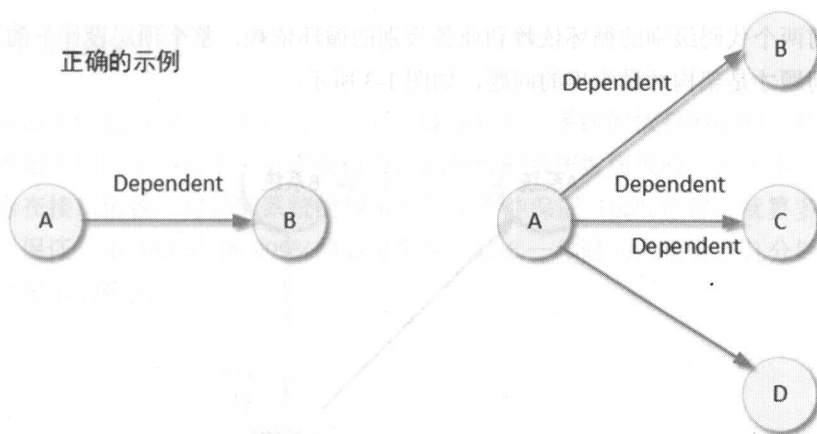


图 1-1 正常的系统间依赖

笔者曾参与开发一款软件，客户方曾经提过这样一个业务需求：货运系统中在创建新的“发车单”时，必须选择空闲的司机和空闲的货车（当然货车类型是要判断的）。空闲司机和空闲货车缺少任何一样都不能完成“发车单”的创建。但同时为了记录某辆货车上一次对应的“发车单”，客户要求只能在创建新的发车单后，货车才能解除之前的“发车单”绑定关系，变成“空闲货车”，如图 1-2 所示。

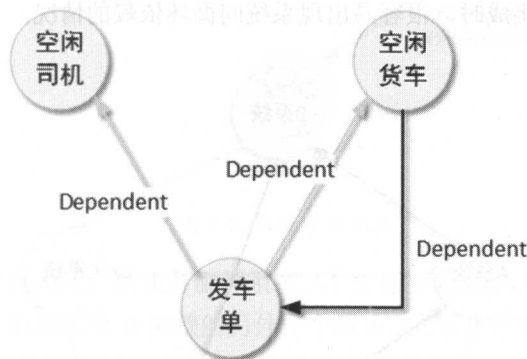


图 1-2 业务间循环依赖

如果按照客户的原始需求，则肯定会出现业务间的循环依赖问题。如果只有在完成新的“发车单”创建后，货车才能解除和之前“发车单”的绑定关系，那么新建“发车单”时，“空闲的货车”从哪里来呢？客户方不懂技术，是我们在需求调研阶段遇到的算一个问题的问

题，但关键看需求人员从哪个方面着手向用户解释，引导用户对需求逻辑进行分析。不一定要用技术语言直接告诉用户，他的需求在技术层面上不符合逻辑。

而比起前两个代码级别的循环依赖和业务级别的循环依赖，整个顶层设计下的若干子系统的循环依赖问题才是架构师最头疼的问题，如图 1-3 所示。

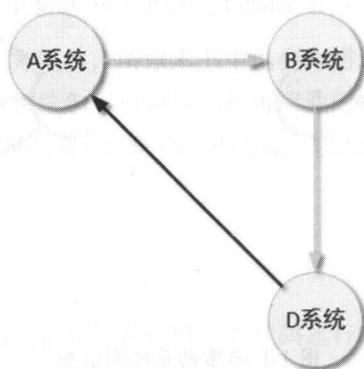


图 1-3 三个系统的循环依赖

在系统数量还没有达到一定数量时（通常来说这个阈值为 4），系统间的循环依赖最可能是由业务人员/技术人员无意造成的。这时，系统间的依赖关系还处于一个可控级别，即使出现系统间循环依赖的情况，技术团队/业务团队也可以快速进行纠正。但是，当参与集成的业务系统数量超过可控制的阈值数量后，这个检查和纠正工作就不再是人工可及的范围了。如图 1-4 所示的五个系统进行集成时，很容易出现系统间循环依赖的情况。

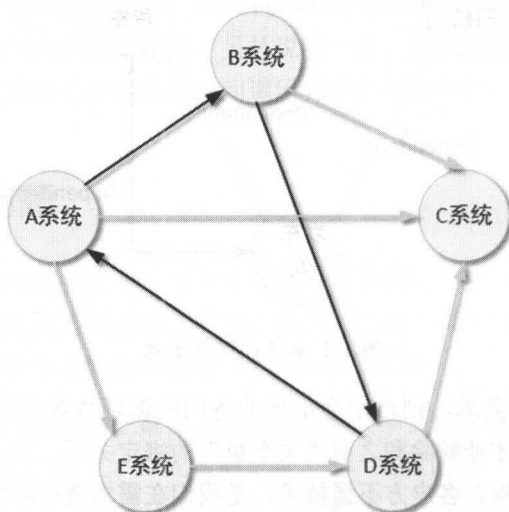


图 1-4 五个系统间循环依赖

## 1.5 系统雪崩问题

当系统出现性能问题、可用性问题、异常处理问题、系统间依赖问题时，技术人员往往不仅要担心问题本身带来的后果，更需要担心由此导致的系统雪崩效应。在大中型业务系统中，系统间都存在接口依赖，对外部系统的调用过程又不能保证 100% 正常，就算多个被调用的系统都声称能保证一年 365 天 99.999% 的高可用性，那么一年时间内也会有几分钟时间处于不可用状态，如图 1-5 所示。

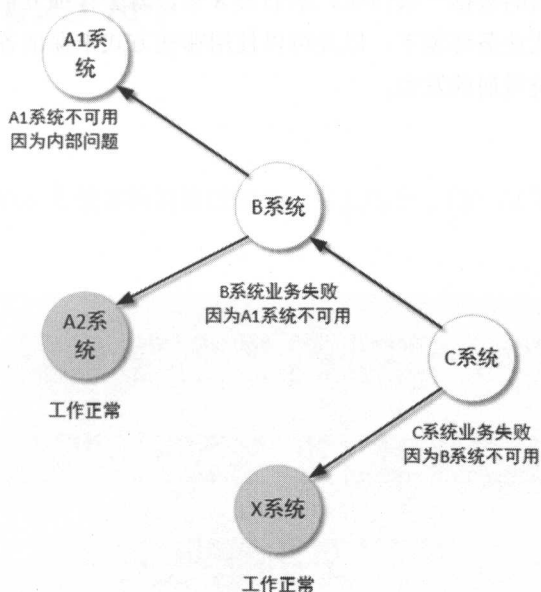


图 1-5 系统雪崩效应

由于系统间存在依赖关系，当业务服务系统 A1 变得不可用时，依赖它的另一个系统 B 的服务也会变得不可用（虽然系统 B 依赖的另外一个服务 A2 还是可用状态，但这不满足充要条件），当系统 B 的服务不可用时，依赖系统 B 工作的服务系统 C 也变得不可用……最后技术人员发现的现象就是，虽然 C 系统一直在报 B 系统的接口调用失败的错误，但是经过排查发现 B 系统工作是正常的，真正出问题的是 A1 系统。这就是一个典型的系统雪崩状态，但还不是最严重的系统雪崩，所以我们需要在本书中讨论这一类型问题的解决思路。关于分布式业务异常的处理和防止系统雪崩的有效方案，将会在本书第三部分第 7 章中讨论。

当然还有一类最常见的问题，就是开发人员写代码时突然“断片”留下的低级问题。由于单元测试覆盖率的缺失，这类问题又没有很好地在自测和集成测试步骤中被发现，最终被发布到上线环境。这类问题往往在发现时就已经造成了严重的后果，例如没有调用出库功能接口，

导致车队一直收不到装车通知。要尽可能避免这类问题最好的办法是优化技术团队的版本发布流程、优化自动测试方案，将问题扼杀在线下。类似这样开发人员所犯的低级错误并不是本书主要的讨论内容，本书主要针对线上环境遇到的技术问题，基于利用已有的成熟的第三方技术组件的思路，以解决各个子系统性能问题、高可用性问题为核心内容，为各位读者提供帮助。例如笔者最初在电信行业的一些公司工作过，技术人员经常使用分布式事务来保证两个业务系统的数据一致性。后来所有的业务都开始向互联网业务转型时，笔者向这些技术团队建议改用RPC的方式完成系统间业务调用，但笔者经常会被问到这样一个问题，如果不使用分布式事务，那么如何保证几个系统间的数据一致性呢？本书第8章，就会详细分析为什么分布式事务不适合使用在高并发的分布式业务环境下，以及可以使用哪些方式来保证各业务系统间的数据一致性，同时又如何防止系统雪崩的发生。

## 第2章 业务系统分解

图 2-1 中描述了 Web 系统架构的组成部分。并且给出了每一层常用的技术组件/服务实现。需要注意以下几点。



图 2-1 业务系统分层架构



- 系统架构是灵活的，根据需求的不同，不一定每一层的技术都需要使用。例如：一些简单的 CRM 系统可能在初期并不需要 K-V 的缓存组件；一些系统因访问量不大，并且可能只有一个服务节点存在，所以也暂时不需要引入负载均衡层。
- 另外请注意，图 2-1 中的业务系统间通信层并没有加入传统的 HTTP 请求方式，因为本书更看重工作在传输层上的各种 -RPC 协议实现在高并发系统间完成业务过程所起的作用。不过目前还是有很多服务间调用采用应用层协议完成系统间调用，例如 Spring Cloud 微服务框架就主要采用 RESTful HTTP 完成已注册服务的调用，其好处在于应用层协议的通用性和简易性。所以可以更多地使用在各种客户端（Web、iOS、Android 等）到服务器端的请求调用过程。
- 我们把业务编码中常使用的缓存系统归入到数据存储层，是因为类似于 Redis 这样的 K-V 存储系统，从本质上讲是一种键值数据库，并且这些内存数据库还可以在技术人员的配置下拥有持久化存储的功能。
- 还有一点需要注意的是，图 2-1 中每层之间实际上不存在绝对的联系（例如负载层一定会把请求转送到业务层，这样的必然性是不存在的），在通常情况下各层是可以跨越访问的。举例说明：如果 HTTP 访问的是一张图片资源，则负载层不会把请求送到业务层，而是直接到部署的分布式文件系统上寻找图片资源并返回。再比如负载层技术还可以直接为数据库层（例如 MySQL 数据库）提供负载均衡服务，这就是负载层直接与数据存储层进行合作的情况。

## 2.1 负载层技术

实际上负载均衡的概念很广泛，所述的过程是将来源于外部的处理压力通过某种规律/手段分摊到内部的各个节点上处理。在日常生活中，我们随时随地在与负载均衡技术打交道，例如：上下班高峰期的车流量引导、民航空管局的航空流量管制、银行柜台的叫号系统等。

这里所说的负载分配层，是单指利用软件实现的计算机系统上的狭义负载均衡。一个大型（日 PV 一亿多）、中型（日 PV 一千万多）业务系统，是不可能只有一个业务服务节点提供服务的，一般是由多个节点同时进行某一个相同业务的服务，所以需要根据业务形态设计一种架构方式，将来自外部客户端的业务请求分担到每一个可用的业务节点上。如图 2-2 所示。



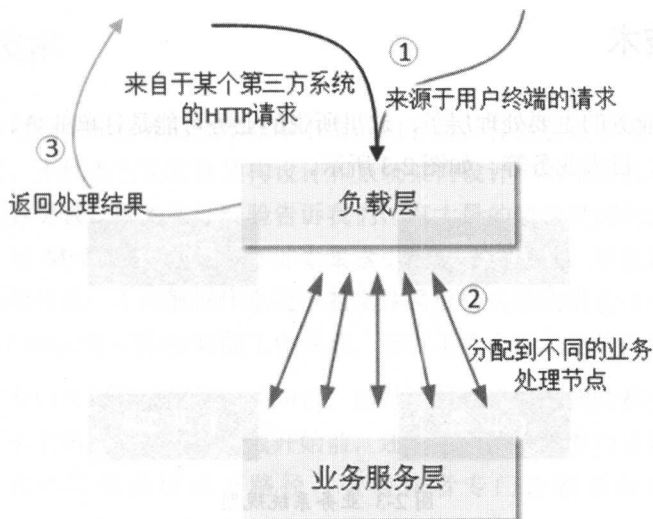


图 2-2 负载层工作过程

- 这里的“用户终端”不只包括类似于 PC、Android 手机、平板电脑这样的终端设备。还包括向服务节点发送请求的任何其他服务节点，所以“用户终端”还可能是某款业务服务系统，也还可能是某款数据分析系统，等等。
- 分配到不同的业务处理节点有两个层面的含义：第一个层面是说，多次同一种类型的请求可以分配到同一个业务系统的不同节点上进行处理；另一个层面是说“用户终端”多次不同类型的请求，会分派到不同的业务系统的不同节点上进行处理。例如：如果某一个 HTTP 请求是请求一张图片，那么负载层会直接到图片存储介质上寻找相应的图片；如果某一个 HTTP 请求是提交一张订单，那么负载层会根据规则将这张订单提交发送到指定的“订单业务系统”的各个节点上。

不同的业务需求，使用的负载方案也是不同的，这就非常考验软件架构师的方案选择能力。例如 Nginx 只能处理 TCP/IP 协议之上的应用层 HTTP 协议，如果要处理 TCP/IP 协议，则要安装第三方的 TCP-Proxy-Module 模块。更好的直接在 TCP/IP 层负载的方案，是使用 HAProxy。随后本书将详细介绍一些常用的负载层架构方案及这些方案的变形。它们包括（但不限于）：

- 独立的 Nginx 负载或 HAProxy 方案
- LVS (DR) + Nginx 方案
- DNS 轮询 + LVS + Nginx 方案
- 智能 DNS (DNS 路由) + LVS + Nginx 方案

## 2.2 业务层技术

这层就是核心业务的主要处理层了，这里所说的业务可能是订单业务、施工管理业务、诊疗业务、付款业务、日志业务等。如图 2-3 所示。



图 2-3 业务系统规划

很明显，在中大型业务系统中，这些业务是不可能独立存在的，一般的设计要求都会涉及子系统间脱耦：即 X1 系统除了知晓底层支撑系统的存在（例如用户权限系统），并不需要知道和它逻辑对等的 X2 系统的存在就可以工作。这种情况下要完成一个较复杂业务，子系统间调用就是必不可少的：例如，A 业务在处理成功后，会调用 B 业务进行执行；A 业务在处理失败后，会调用 C 业务进行执行；又或者 A 业务和 D 业务在某种情况下是不可分割的整体，只有同时成功才成功，若其中有一个失败则整个业务过程都会失败。如图 2-4 所示。

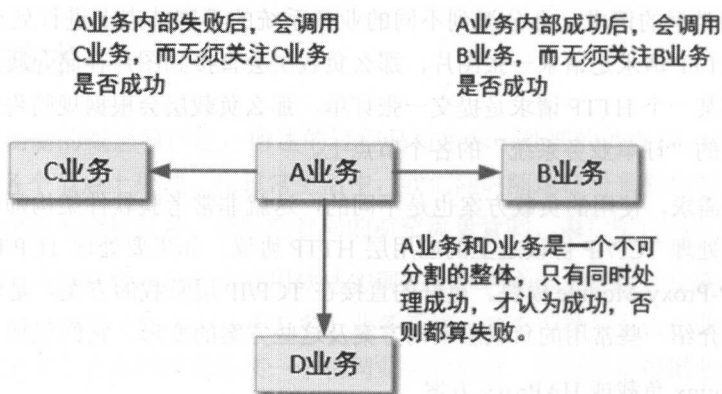


图 2-4 业务系统协作

在随后的章节中，我们将以 Apache Thrift RPC 技术，保证消息可达性的消息队列技术的原理和使用方式，来讲解业务通信层技术，特别是业务通信层的技术选型注意事项。还会特别介绍在业务层进行多个业务调用时的异常处理方案，包括事务补偿逆操作和错误熔断操作。

## 2.3 存储层技术

相比负载均衡层技术和业务层技术在本书中的内容量，存储层技术确实就要少得多了。但是不讲解相关知识，不代表它在软件架构设计和系统架构设计中的重要性不如前两者，相反它的重要性要比前两者更甚，因为实际经验告诉我们，有大量的错误是因为没有处理好系统存储而造成的，例如，对 MySQL 聚集索引和非聚集索引理解不够透彻，导致常常出现 Table Scan 拖慢整个顶层系统的性能；不理解操作系统下的文件系统格式导致错选 EXT3、EXT4、XFS；不清楚 Linux Page Cache 对文件读/写的工作原理，所以不能合理利用代码进行批量写。

虽然本书没有专门介绍存储层的相关知识，但是本书在讲解示例时都会附带说明其中涉及的存储层支持，在本书第四部分场景实战开始前，还用了一些篇幅专门讲解实战中必要的系统存储层知识。读者还可以通过以下路径，查看作者专门讲解系统存储的网络资源：  
<http://blog.csdn.net/column/details/12844.html>。

## 第二部分

# 负载层技术与设计

---

本部分内容重点向读者讲解负载层使用的技术原理、流行的产品和产品的安装调试过程。并通过一个实际例子将这些知识点贯穿起来。由于将要讲述的产品其知识点比较多，所以占用的章节比较多。另外由于本书针对的目标读者定位问题，所以关于应用软件的安装过程只会进行粗略介绍，其目的只是提醒读者在实际安装过程中可能会遇到的问题。本部分的重点还是软件的工作原理讲解和性能调整。

## 第 3 章

# Nginx 技术

Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，并在一个 BSD-like 协议下发行（一种开源协议）。Nginx 由俄罗斯的程序设计师 Igor Sysoev 所开发，最初供俄罗斯大型的入口网站及搜索引擎 Rambler（俄文：Рамблер）使用。其特点是占有内存少，并发能力强，事实上 Nginx 的并发能力确实在同类型的网页服务器中表现较好，国内使用 Nginx 的网站包括（但不限于）：百度、京东、新浪、网易、腾讯等。

## 3.1 Nginx 中的基本技术理论

在正式介绍 Nginx 之前，我们先介绍几个关键的算法与工作原理，这些算法在 Nginx 中，或者说在整个负载层技术乃至整个软件系统领域中都起着至关重要的作用。如果读者还不了解这些算法在 Nginx 中所起的作用，请不要着急，后文将依次说明它们的作用。

### 3.1.1 一致性 Hash 算法

一致性 Hash 算法是现代系统架构中最关键的算法之一，它在分布式计算系统、分布式存储系统、数据分析等众多领域中被广泛应用。针对本书内容，读者可以在负载均衡层、服务调用层和数据存储层中找到它的身影。如图 3-1 所示。

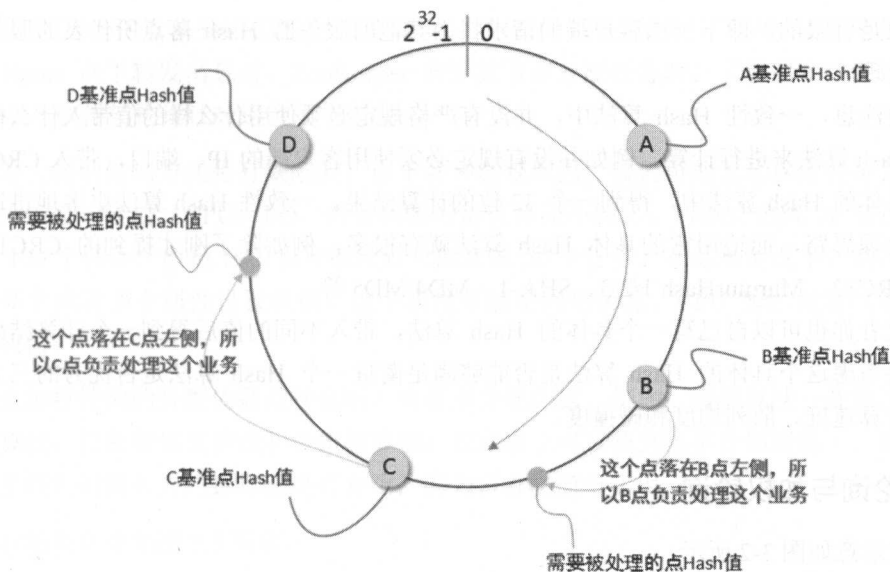


图 3-1 一致性 Hash 算法

- 一致性 Hash 算法的关键思路在于它能够根据不同的属性数据，生成一串不相同的 Hash 值，并且能够将这个 Hash 值转换为 0 至  $2^{32}-1$  范围的整数（或者有可排列依据的其他数据类型），形成图 3-1 中的圆环。请注意这个圆环是一个抽象结构，在实际编程中需要使用各种数据结构进行模拟，例如在数据库中间件 MyCat 中，开发人员就使用了红黑树来形成这个抽象的环；Ceph 文件系统在若干个 PG 中选择一个 PG 进行 Object 存储时，也使用一致性 Hash 算法完成。
- 一台服务器的某个或者某一些属性当然可以进行 Hash 计算（参与计算的属性通常是这个服务器的 IP 地址和开放端口），并且根据计算分布在这个圆环上的某一个点。也就是图中圆环上的 A 点、B 点、C 点或者 D 点（下文称为服务器 Hash 落点）。
- 一个“处理请求”当然也可以根据这个请求的某一个或者某一些属性进行 Hash 计算（参与计算的属性可以是这个请求的 IP、端口、Cookie 值、URL 值或请求时间等），并且根据计算记过分布在这个圆环上的某一个点上。也就是图 3-1 中圆环上的浅灰色小点（下文称为处理请求落点）。
- 我们约定“处理请求”经过 Hash 计算后，落在 A 点左侧和 B 点右侧，则表示这些“处理请求”都由 A 点所代表的服务器进行处理，这样就完全解决了“谁来处理”的问题。在服务器 Hash 落点稳定存在的前提下，来自于同一个 IP 和端口的“处理请求”经过 Hash 计算后的数值，落在圆环的位置都是一样的，这就保证了服务处理映射的稳定性。
- 当某一个服务器 Hash 计算结果的落点由于某种原因下线，其所影响到的处理请求落点

也是有限的。即下一次客户端的请求将由其他的服务器 Hash 落点所代表的服务器进行处理。

- 请注意，一致性 Hash 算法中，并没有严格规定必须使用什么样的值带入什么样具体的 Hash 算法来进行计算。例如并没有规定必须使用客户端的 IP、端口，带入 CRC16 这个具体的 Hash 算法中，得到一个 32 位的计算结果。一致性 Hash 算法更多地讲述了一个处理思路，而适用它的具体 Hash 算法就有很多，例如除了刚才提到的 CRC16，还有 CRC32、MurmurHash 1/2/3、SHA-1、MD4/MD5 等。
- 或者你也可以自己写一个具体的 Hash 算法，带入不同的值后得到一个计算结果。不过要考虑这个具体的 Hash 算法是否能够满足衡量一个 Hash 算法是否优秀的三大标准：计算速度、散列均度和碰撞度。

3.1.2 轮询与加权轮询

轮询示意如图 3-2 所示。

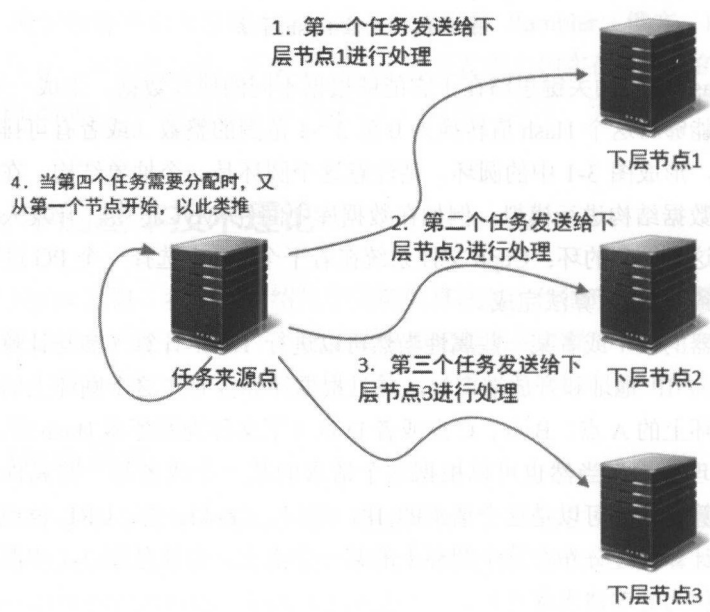


图 3-2 轮询

- 当有任务需要传递到下层节点进行处理时，任务来源点会按照一个固定的顺序，将任务依次分配给下层节点，如果下层可用的节点数量为 X，那么第 N 个任务的分配规则为：

$$\text{目标节点} = (N \bmod X) + 1$$

- 轮询处理在很多架构思想中都有体现：DNS 解析多 IP 时、LVS 向下转发消息时、Nginx 向下转发消息时、ZooKeeper 向计算节点分配任务时。了解基本的轮询过程有助于在软件架构设计时进行思想借鉴。

但是上面的轮询方式是有缺陷的，由于各种客观原因，我们可能无法保证任务处理节点的处理能力都是一样的（CPU、I/O、内存频率等都有可能不同）。所以会出现 A 节点业务能同时处理 100 个任务，但是 B 节点可能同时只能处理 50 个任务。在这种情况下，则需要依据下层节点某个或者多个属性设置权值。这个属性可能是网络带宽、CPU 繁忙程度或者就是一个固定的权值。

那么加权轮询的分配依据是什么呢？有很多分配依据，例如：概率算法（此算法中包括蒙特卡罗算法、拉斯维加斯算法和舍伍德算法，在网络上可以找到很多介绍资料）、最大公约数法。这里我们对最大公约数算法进行介绍，因为该方法简单实用。

加权轮询示意如图 3-3 所示。

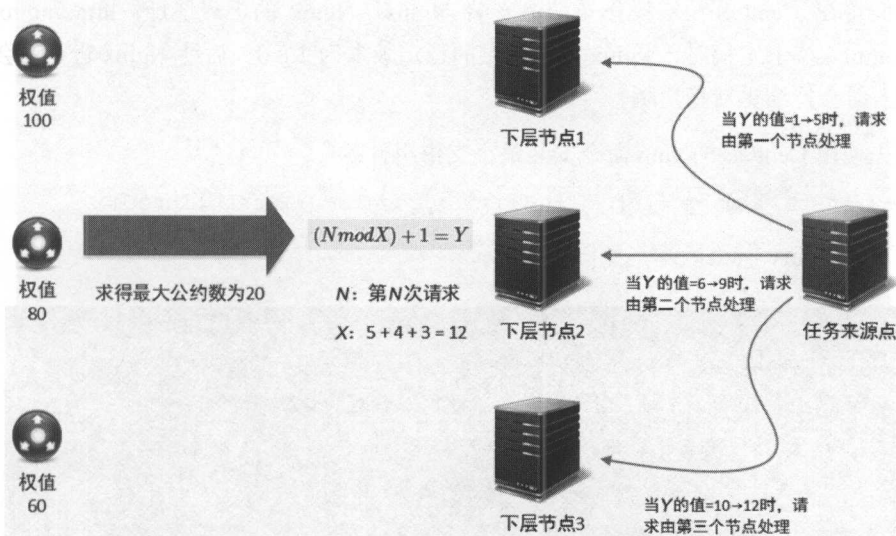


图 3-3 加权轮询

请注意图 3-3 中的“→”符号表示取值范围，例如取值为 1、2、3、4、5。

- 在加权轮询算法中，首先按照某种规则计算得到每个处理节点的权值。上文已经说到计算规则可能是这个服务节点的 CPU 利用率、网络占用情况或者在配置文件中的固定权重。
- 求这些权值的最大公约数，在图 3-3 中三个节点的权值分别是 100、80、60。那么求得



的最大公约数就是 20（如果你忘记了最大公约数的定义，请自行复习）。那么这三个节点的被除结果分别是 5、4、3，求和值为 12。

- 得到以上的计算结果，就可以开始进行请求分配了，公式同样为：

$$(N \bmod X) + 1 = Y$$

其中  $N$  表示当前的第  $N$  次任务； $X$  表示整除后的求和结果； $Y$  为处理节点。

加权轮询是轮询方案的补充，通过将处理节点的属性转换成权值可以有效地描述处理节点的处理能力，实现更科学的处理任务分配。加权轮询的关键在于加权算法，最大公约数算法简单实用、定位效率高。

## 3.2 Nginx 的安装和使用

本节介绍 Nginx 的安装过程，Nginx 的安装很简单，所以本节只会花较少的篇幅进行粗略介绍。我们将在 CentOS 6.X 操作系统上安装 Nginx。Nginx 的下载地址：<http://nginx.org/en/download.html>。本书定稿时，Nginx 官方提供的稳定版本为 1.8.0。后续 Nginx 肯定还会不断升级，官网上面会持续更新稳定版。

- 首先使用 CentOS 的 yum 命令安装最小支撑组件：

```
yum -y install make zlib zlib-devel gcc gcc-c++ ssh libtool
```

接着下载 Nginx 1.8.0 版本，如图 3-4 所示。

```
[root@localhost ~]# wget http://nginx.org/download/nginx-1.8.0.tar.gz
--2015-07-04 20:59:36-- http://nginx.org/download/nginx-1.8.0.tar.gz
正在解析主机 nginx.org... 206.251.255.63, 2606:7100:1:69::3f
正在连接 nginx.org[206.251.255.63]:80... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度：832104 (813K) [application/octet-stream]
正在保存至：'nginx-1.8.0.tar.gz'

26% [=====>] 220,039 68.8K/s eta(英国中部时
27% [=====>] 228,679 66.8K/s eta(英国中部时
29% [=====>] 241,639 64.8K/s eta(英国中部时
29% [=====>] 243,079 61.4K/s eta(英国中部时
29% [=====>] 244,519 57.8K/s eta(英国中部时
29% [=====>] 247,399 55.7K/s eta(英国中部时
30% [=====>] 253,159 54.4K/s eta(英国中部时
31% [=====>] 253,220 54.2K/s eta(英国中部时
```

图 3-4 安装过程 1

- 解压 Nginx 的 tar 文件到指定位置（或者当前位置），如图 3-5 所示。

```
[root@localhost ~]# tar -zxvf ./nginx-1.8.0.tar.gz
nginx-1.8.0/
nginx-1.8.0/auto/
nginx-1.8.0/conf/
nginx-1.8.0/contrib/
nginx-1.8.0/src/
nginx-1.8.0/configure
nginx-1.8.0/LICENSE
nginx-1.8.0/README
nginx-1.8.0/html/
nginx-1.8.0/man/
nginx-1.8.0/CHANGES.ru
nginx-1.8.0/CHANGES
nginx-1.8.0/man/nginx.8
```

图 3-5 安装过程 2

- 进行源码编译安装，如图 3-6 所示。

```
[root@localhost ~]# cd ./nginx-1.8.0
[root@localhost nginx-1.8.0]# ./configure --prefix=/usr/nginx-1.8.0
checking for OS
+ Linux 2.6.32-504.16.2.el6.x86_64 x86_64
checking for C compiler ... found
+ using GNU C compiler
+ gcc version: 4.4.7 20120313 (Red Hat 4.4.7-11) (GCC)
checking for gcc -pipe switch ... found
checking for gcc builtin atomic operations ... found
checking for C99 variadic macros ... found
checking for gcc variadic macros ... found
checking for PCRE library ... not found
checking for PCRE library in /usr/local/ ... not found
checking for PCRE library in /usr/include/pcr/ ... not found
checking for PCRE library in /usr/pkg/ ... not found
checking for PCRE library in /opt/local/ ... not found

./configure: error: the HTTP rewrite module requires the PCRE library.
You can either disable the module by using --without-http_rewrite_module
option, or install the PCRE library into the system, or build the PCRE library
statically from the source with nginx by using --with-pcre=<path> option.
```

图 3-6 安装过程 3

这时我们看到编译检查报错了，报错内容写得很清楚：为了支持 HTTP 重写模块，Nginx 需要安装 PCRElib。那么我们到 <http://ncu.dl.sourceforge.net/project/pcre/pcre/8.37/pcre-8.37.tar.gz> 下载一个稳定的 PCRE 版本编译安装即可（不一定是 8.37 版本）。

- 安装 PCRE 后，再次进行编译安装，如图 3-7 所示。

```
./configure --prefix=/usr/nginx-1.8.0
make && make install
```

```

[root@vml nginx-1.8.0]# ./configure --prefix=/usr/nginx-1.8.0
checking for OS
+ Linux 2.6.32-431.el6.x86_64 x86_64
checking for C compiler ... found
+ using GNU C compiler
+ gcc version: 4.4.7 20120313 (Red Hat 4.4.7-4) (GCC)
checking for gcc -pipe switch ... found
checking for gcc builtin atomic operations ... found
checking for C99 variadic macros ... found
checking for gcc variadic macros ... found
checking for unistd.h ... found
checking for inttypes.h ... found
checking for limits.h ... found
checking for sys/filio.h ... not found
Configuration summary
+ using system PCRE library
+ OpenSSL library is not used
+ md5: using system crypto library
+ sha1: using system crypto library
+ using system zlib library

[root@vml nginx-1.8.0]# make && make install
make -f objs/Makefile
make[1]: Entering directory `/root/nginx-1.8.0'
cc -c -pipe -O -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I src/os/unix -I objs \
-o objs/src/core/nginx.o \
src/core/nginx.c
cc -c -pipe -O -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I src/os/unix -I objs \
-o objs/src/core/nginx_log.o \
src/core/nginx_log.c
cc -c -pipe -O -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I src/os/unix -I objs \

```

图 3-7 安装过程 4

整个验证、编译、安装过程不应该报任何错误。如果你使用 `prefix` 设置了安装目标目录，那么可能还需要在 `/etc/profile` 文件中设置环境变量。

下面介绍几个 Nginx 常用的命令，如果可以正常使用这些命令，则说明 Nginx 已经安装成功了。

- `nginx`: 直接在命令行键入 `nginx`，就可以启动 Nginx。
- `nginx-t`: 检查配置文件是否正确。这个命令可以检查 `nginx.conf` 配置文件的格式、语法是否正确。如果配置文件存在错误，则会出现相应提示；如果 `nginx.conf` 文件正确，也会出现相应的成功提示。
- `nginx -s reload`: 重加载/重启 Nginx——以新的 `nginx.conf` 配置文件定义。
- `nginx -s stop`: 停止 Nginx。

### 3.3 Nginx 的重要配置讲解

3.2 节的安装过程是不是非常简单？已经有过 Nginx 使用经验的读者甚至可以忽略上一小

节的阅读。如果你是按照本书的描述方式安装的 Nginx，那么 Nginx 的主配置文件在：  
/usr/nginx-1.8.0/conf/nginx.conf；如果你在编译安装时并没有指定安装目录，那么 Nginx 的主配置文件在：/usr/local/nginx/conf/nginx.conf。当然你还可以在启动 Nginx 时使用 -c 的参数人为指定 Nginx 的配置文件位置（但是这种方式不建议）。

Nginx 在安装完成后，不用更改任何配置信息就可以直接运行。但是很显然这不能满足我们生产环境的要求。这里本书重新整理了 Nginx 的配置文件，将其分块，并向读者讲解其中重要的配置信息：

```
#=====以下是全局配置项
#指定运行 Nginx 的用户和用户组，
#默认情况下该选项关闭（关闭的情况下使用的用户就是 nobody）
#user  nobody nobody;
#运行 Nginx 的进程数量，后文详细讲解
worker_processes 1;
#Nginx 运行错误的日志存放位置。当然你还可以指定错误级别
#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;
#指定主进程 id 文件的存放位置，虽然 worker_processes != 1 的情况下
#会有很多进程，管理进程却只有一个
#pid        logs/nginx.pid;
events {
    #每一个进程可同时建立的连接数量，后文详细讲解
    worker_connections 1024;
    #连接使用的网络 I/O 模型，
    #可以采用[kqueue rtsig epoll select poll eventport ]，后文详细讲解
    use  epoll;
}
http {
    #=====以下是 Nginx 后端服务配置项
    upstream backendserver1 {
        #Nginx 向后端服务器分配请求任务的方式，默认为轮询
        #如果指定了 ip_hash，就是 Hash 算法
        #ip_hash
        #后端服务器 ip:port，如果有多个服务节点，这里就配置多个
        server 192.168.220.131:8080;
        server 192.168.220.132:8080;
        #backup 表示，这是一个备份节点，只有当所有节点失效后
        #Nginx 才会往这个节点分配请求任务
        #server 192.168.220.133:8080 backup;
        #weight，固定权重，还记得我们上文提到的加权轮询方式吧
        #server 192.168.220.134:8080 weight=100;
```



```

}
#====以下是 HTTP 协议主配置
#安装 Nginx 后, 在 conf 目录下除了 nginx.conf 主配置文件, 还有很多模板配置文件,
#这里就是导入这些模板文件
include      mime.types;
#HTTP 核心模块指令, 这里设定默认类型为二进制流, 也就是当文件类型未定义时使用这种方式
default_type application/octet-stream;
#日志格式
#log_format main '$remote_addr - $remote_user [$time_local] "$request" '
#               '$status $body_bytes_sent "$http_referer" '
#               '"$http_user_agent" "$http_x_forwarded_for"';

#日志文件存放的位置
#access_log logs/access.log main;
#sendfile 规则开启
sendfile     on;
#指定一个连接的等待时间(单位秒), 如果超过等待时间, 则连接就会断掉。注意一定要设
#置, 否则高并发情况下会产生性能问题
keepalive_timeout 65;
#开启数据压缩, 后文详细介绍
gzip on;
#====以下是一个服务实例的配置
server {
    #这个代理实例的监听端口
    listen      80;
    #server_name 取个唯一的实例名都要想半天?
    server_name localhost;
    #编码格式
    charset utf-8;
    #access_log logs/host.access.log main;
    #location 将按照规则分流满足条件的 URL。
    #"location /"你可以理解为“默认分流位置”。
    location / {
        #root 目录, 这个 html 表示 Nginx 主安装目录下的“html”目录。
        root    html;
        #目录中的默认展示页面
        index   index.html index.htm;
    }
    #location 支持正则表达式, “~” 表示匹配正则表达式
    location ~ ^/business/ {
        #方向代理。后文详细讲解
        proxy_pass http://backendserver1;
    }
    #redirect server error pages to the static page /50x.html
    #error_page 404                /404.html;

```

```

error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root html;
}

```

## 3.4 Nginx 的重要设置

### 3.4.1 use [ kqueue | rtsig | epoll | select | poll ]

科技在发展，社会在进步，满足摩尔定义的 IT 行业更是这样。以前所有的网络 I/O 模型都是阻塞式的（Blocking I/O）。也就是说一个 TCP 连接线程在发出请求后，代码就不会再往下执行了，直到得到远端的 Response 为止；服务器端也一样，在处理完某一个客户端的请求之前，其他客户端的请求都会等待。这种处理方式使客户端和服务端端的通信性能大打折扣。

从问题的表面现象看，多线程技术似乎能够解决这个问题：一个线程处理不完，可以再开多个线程来处理。但是多线程是有很多局限性的：

- 创建一个线程会消耗有限的资源。以 JVM（Java 虚拟机）为例，创建一个新的线程，JVM 会单独开放一定的栈内存空间（通过-Xss 参数可设置），虽然栈内存不受-Xmx 和-Xms 两个参数影响，但是可以说明线程的创建是需要消耗额外资源的。
- 多线程工作时，计算机的 CPU 会耗费大量的资源让多线程在不同的状态下进行切换。在后续的章节中还会依据这样的原理直接写一段相当规模的线程在挂起、就绪状态不停地切换操作，来玩死 CPU。
- 在 Linux 操作系统下，单个用户能够创建的线程和进程总数、整个操作系统能够创建的线程总数都是有限的。通过 limit -a 命令，可以查看相关的内核参数。
- 所以依靠线程来解决 BIO 的问题是不靠谱的，只能起到缓解处理并行请求压力的作用。如果一次并发 10 万个处理请求的问题，则是不可能在计算机上同时创建 10 万个线程来解决的。

基于上面的描述，NIO（No Blocking I/O，本书定位于多路复用 I/O 模型，后继章节会详细介绍）技术就这样诞生了。依靠 Linux 操作系统下的多路复用机制，单个线程可以同时处理多个请求，并在处理后回调相关的远程事件。根据 NIO 实现机制的不同，技术名称也就不同了。

epoll、kqueue：NIO 的实现，其中 epoll 是 poll 的改进实现，在 Linux 环境下可以使用，但限于 Linux 2.6 及以上版本。kqueue 主要在 BSD 中使用。

3.4.2 worker\_processes 和 worker\_connections

**worker\_processes:** 这个属性表示操作系统启动多少个工作进程运行 Nginx。注意是工作进程，不是有多少个 Nginx 工程。在 Nginx 运行时，会启动两种进程，一种是主进程 master process；一种是工作进程 worker process。例如笔者在 2.3 节给出的配置文件中将 worker\_processes 设置为 4，启动 Nginx 后，使用进程查看命令观察名字叫作 Nginx 的进程信息，读者会看到如图 3-8 所示结果。

```
[root@vml ~]# ps -elf | grep nginx
1 S root      1615      1  0  80   0 -  6069  rt_sig  04:21 ?        00:00:00 nginx: master process /usr/
5 S nobody   2596   1615  0  80   0 -  6174  ep_pol  04:28 ?        00:00:00 nginx: worker process
5 S nobody   2597   1615  0  80   0 -  6174  ep_pol  04:28 ?        00:00:00 nginx: worker process
5 S nobody   2598   1615  0  80   0 -  6174  ep_pol  04:28 ?        00:00:00 nginx: worker process
5 S nobody   2599   1615  0  80   0 -  6174  ep_pol  04:28 ?        00:00:00 nginx: worker process
0 S root      2663   2397  0  80   0 - 25816  pipe_w  04:28 pts/0    00:00:00 grep nginx
```

图 3-8 Nginx 进程信息

以上命令示例中看到 1 个 Nginx 主进程 master process，还有 4 个工作进程 worker process。主进程负责监控端口，协调工作进程的工作状态，分配工作任务，工作进程负责进行任务处理。一般这个参数要和操作系统的 CPU 内核数成倍数。

**worker\_connections:** 这个属性是指单个工作进程可以允许同时建立外部连接的数量，无论这个连接是 Nginx 外部建立的，还是 Nginx 内部建立的。这里需要注意的是，一个工作进程建立一个连接后，进程将打开一个文件副本。所以这个数量还与操作系统设定的进程最大可打开的文件副本数有关。大概有一半网络资料告诉了读者这个事实，并要求在修改 worker\_connections 属性时，一定要使用 ulimit -n 修改操作系统对进程最大文件数的限制，但是这样更改只能在当次用户的当次 shell 会话中起作用，并不是永久的。如果继续搜索，则会发现另外还有 30% 的网络资料告诉你，要想使“进程最大可打开的文件数”永久有效，还需要修改/etc/security/limits.conf 这个主配置文件，但是应该如何正确检查“进程的最大可打开文件”却没有说。所以本书为读者详细描述了正确的设置方式。

1. 更改操作系统“进程最大可打开文件数”的设置

首先需要操作系统的 root 权限，然后需要修改 limits.conf 主配置文件：

```
> vim /etc/security/limits.conf

在主配置文件的最后加入下面两句：

* soft nofile 65535
* hard nofile 65535
```

注意“\*”是要加到文件里面的。以上在 limits.conf 文件中添加的两句话的含义是 soft（应

用软件) 级别限制的最大可打开文件数的限制, **hard** 表示操作系统级别限制的最大可打开文件数的限制, “\*” 表示所有用户都生效。保存这个文件 (只有 **root** 用户能够有权限)。

保存这个文件后, 配置是不会马上生效的, 为了保证本次 **shell** 会话中的配置马上有效, 我们需要通过 **ulimit** 命令更改本次的 **shell** 会话设置 (或者是重启 Linux 操作系统):

```
> ulimit -n 65535
```

执行命令后, 配置马上生效。可以用 **ulimit -a** 查看目前会话中的所有核心配置:

```
> ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
scheduling priority (-e) 0
file size (blocks, -f) unlimited
pending signals (-i) 7746
max locked memory (kbytes, -l) 64
max memory size (kbytes, -m) unlimited
open files (-n) 65535
pipe size (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority (-r) 0
stack size (kbytes, -s) 10240
cpu time (seconds, -t) unlimited
max user processes (-u) 7746
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited
```

请注意, “open files” 这一项变成了 65535, 说明设置已经生效了。

## 2. 更改 Nginx 软件“进程最大可打开文件数”的设置

刚才更改的只是操作系统级别的“进程最大可打开文件”的限制。对 Nginx 来说, 还要对这个软件进行更改。打开 **nginx.conf** 主配置文件。需要配合 **worker\_rlimit\_nofile** 属性。如下:

```
#.....nginx.conf 文件中相关的配置内容
user root root;
worker_processes 4;
worker_rlimit_nofile 65535;
#error_log logs/error.log;
#error_log logs/error.log notice;
#error_log logs/error.log info;
#pid logs/nginx.pid;
events {
    use epoll;
    worker_connections 65535;
```



```
}
```

这里只给出了 `nginx.conf` 文件中受影响的部分配置内容，其他的配置代码和主题无关也就不在这里赘述了。请注意代码行中加粗的两个配置项，请一定将两个属性全部配置。配置完成后，请通过 `nginx -s reload` 命令重新启动 Nginx。

3. 验证 Nginx “进程最大可打开文件数” 是否起作用

那么，我们如何来验证配置是否起作用了呢？在 Linux 系统中，所有的进程都会有一个临时的核心配置文件描述，存放路径在 `/proc/进程号/limit`。

首先来看一下，没有进行参数优化前的进程配置信息：

```
> ps -elf | grep nginx
1 S root 1594 1 0 80 0 - 6070 rt_sig 05:06 ? 00:00:00 nginx: master process
/usr/nginx-1.8.0/sbin/nginx
5 S root 1596 1594 0 80 0 - 6176 ep_pol 05:06 ? 00:00:00 nginx: worker process
5 S root 1597 1594 0 80 0 - 6176 ep_pol 05:06 ? 00:00:00 nginx: worker process
5 S root 1598 1594 0 80 0 - 6176 ep_pol 05:06 ? 00:00:00 nginx: worker process
5 S root 1599 1594 0 80 0 - 6176 ep_pol 05:06 ? 00:00:00 nginx: worker process
```

可以看到，Nginx 工作进程的进程号是：1596 1597 1598 1599。我们选择其中一个进程，查看其核心配置信息：

```
> cat /proc/1598/limits
```

这样的命令会显示如图 3-9 所示的状态结果。

Limit	Soft Limit	Hard Limit	Units
Max cpu time	unlimited	unlimited	seconds
Max file size	unlimited	unlimited	bytes
Max data size	unlimited	unlimited	bytes
Max stack size	10485760	unlimited	bytes
Max core file size	0	unlimited	bytes
Max resident set	unlimited	unlimited	bytes
Max processes	7746	7746	processes
Max open files	1024	4096	files
Max locked memory	65536	65536	bytes
Max address space	unlimited	unlimited	bytes
Max file locks	unlimited	unlimited	locks
Max pending signals	7746	7746	signals
Max msgqueue size	819200	819200	bytes
Max nice priority	0	0	
Max realtime priority	0	0	
Max realtime timeout	unlimited	unlimited	us

图 3-9 未起作用时

请注意其中的“Max open files”，分别是 1024 和 4096。那么更改配置信息，并重启 Nginx 后，配置信息如图 3-10 所示。

Limit	Soft Limit	Hard Limit	Units
Max cpu time	unlimited	unlimited	seconds
Max file size	unlimited	unlimited	bytes
Max data size	unlimited	unlimited	bytes
Max stack size	10485760	unlimited	bytes
Max core file size	0	unlimited	bytes
Max resident set	unlimited	unlimited	bytes
Max processes	7746	7746	processes
Max open files	65535	65535	files
Max locked memory	65536	65536	bytes
Max address space	unlimited	unlimited	bytes
Max file locks	unlimited	unlimited	locks
Max pending signals	7746	7746	signals
Max msgqueue size	819200	819200	bytes
Max nice priority	0	0	
Max realtime priority	0	0	
Max realtime timeout	unlimited	unlimited	us

图 3-10 起作用后的效果

从以上两段进程配置结果可以看到，“Max open files”属性确实发生了变化，这说明我们的设置成功了。请注意在生产环境下，各位读者一定要确保 Nginx 工作进程的配置信息是经过优化设置的，否则 Nginx 对并发请求的处理能力会大打折扣。

### 3.4.3 max client 的计算方式

这个小节我们主要来说明两个在网络资料上经常提到的公式：

```
max_client = worker_processes * worker_connections
max_client = worker_processes * worker_connections / 4
```

这两个公式分别说明，在 Nginx 充当服务器（例如 Nginx 上面装载 PHP）时，Nginx 可同时承载的连接数量是最大工作线程×每个线程允许的连接数量；当 Nginx 充当反向代理服务时，其可同时承载的连接数量是最大工作线程×每个线程允许的连接数量 / 4。

第一个问题很好理解，关键是第二个问题：为什么会除以 4。有部分资料的答案是：浏览器连接到 Nginx、Nginx 连接到后端服务器、后端服务器再连接到 Nginx、Nginx 最后连接到浏览器，所以需要除以 4，笔者想说的是 TCP 协议是双向全双工协议，为什么需要这样建立连接呢？所以这个说法肯定是错误的。

在 Nginx 官方文档上有这样一句话：

Since a browser opens 2 connections by default to a server and nginx uses the fds (file descriptors) from the same pool to connect to the upstream backend.

翻译成中文的描述就是，浏览器会建立两条连接到 Nginx（注意两条连接都是浏览器建立的），Nginx 也会建立对应的两条连接到后端服务器，这样就是四条连接了。

## 3.5 Nginx 的常用模块

Nginx 通过模块的方式支持很多第三方组件，这些组件可以是其他组织机构开发的，也可以是 Nginx 自带的。本节将介绍几个常用的 Nginx 模块，它们是：gzip 模块、rewrite 模块、健康检查模块、proxy-cache 模块和图片操作模块 `image_filter_module`。由于本书篇幅原因，不可能介绍所有的模块，例如有一个可以和 `jsession` 配合使用的模块 `nginx_upstream_jvm_route` 就是常用模块；还有一个 Nginx 内置的 `proxy-cache` 模块也是常用模块，这个模块的使用本书将在第四部分讲解图片服务实战场景时再介绍其功能和使用方式。

### 3.5.1 gzip 压缩模块

Nginx 对返回给请求端的 HTTP response body 可以进行压缩——通过 gzip 模块。这虽然需要消耗一点 CPU 和内存资源，但是想到其可以将 100KB 的数据量压缩到 60KB 甚至更小进行传输，是否就有一定的吸引力了？HTTP 返回数据进行压缩的功能在很多场景下都适用：

- 如果客户端使用的是 3G/4G 网络，那么 HTTP 流量对于用户来说就是钱。
- 压缩可节约服务器机房的对外带宽，为更多用户服务。按照目前的市场价，良好的机房带宽资源一般在 200 元/Mbps，而服务器方案的压力往往也来自于机房带宽。

笔者的建议是，不要为了节约成本将业务服务和负载层服务放在一台物理服务器上，这样做既影响性能又增加了运维难度。另外需要注意的是，并不是 Nginx 开启了 gzip 功能，HTTP 响应的数据就一定会被压缩：除了满足 Nginx 设置的“需要压缩的 HTTP 格式”，客户端（浏览器）也需要支持 gzip——好消息是目前大多数浏览器和编程语言的 API 都支持 HTTP 压缩。

首先来讲解 Nginx 中的 gzip 的参数设置，然后讲解当开启压缩功能后，HTTP 的交互过程和过程中关键的几个属性。下面首先来看看 Nginx 中开启 gzip 的属性（gzip 的设置放置在 HTTP 主配置区域）：

```
#开启 gzip 压缩服务
gzip on;
#gzip 压缩需要申请临时内存空间，假设前提是压缩后大小是小于等于压缩前的。例如，如果原始
#文件大小为 10KB，那么它超过了 8KB，所以分配的内存是  $8 \times 2 = 16\text{KB}$ ；再例如，原始文件
#大小为 18KB，很明显 16KB 也是不够的，那么按照  $8 \times 2 \times 2 = 32\text{KB}$  的大小申请内存。如果没有
#设置，则默认值是申请跟原始数据相同大小的内存空间去存储 gzip 压缩结果
gzip_buffers 2 8k;
#进行压缩的原始文件的最小值，也就是说如果原始文件小于 5KB，那么就不会进行压缩了
gzip_min_length 5K;
#gzip 压缩基于的 HTTP 协议版本，默认就是 HTTP 1.1
gzip_http_version 1.1;
# gzip 压缩级别 1~9，级别越高压缩率越大，压缩时间也就越长，CPU 越高
```

```
gzip_comp_level 5;
#需要进行 gzip 压缩的 Content-Type 的 Header 的类型。建议 JS、TEXT、CSS、XML、JSON
#格式都要进行压缩；图片就没必要了，GIF、JPGE 文件已经压缩得很好了，就算再压缩，效果也
#不好，而且还耗费 CPU。
gzip_types    text/HTML    text/plain    application/x-javascript    text/css
application/xml;
```

设置完成后，重启 Nginx 即可生效。下面来看看浏览器和服务器进行 gzip 压缩的请求和处理返回过程，如图 3-11 所示。

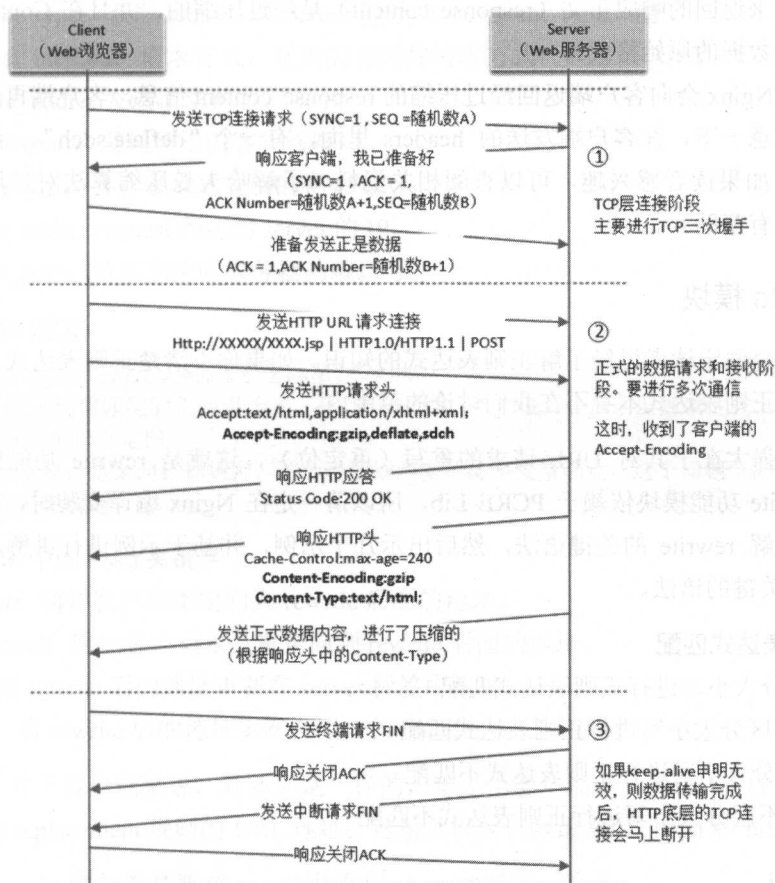


图 3-11 gzip 工作过程

- 整个请求过程中开启 gzip 功能模块和不开启 gzip 功能模块，其 HTTP 的请求和返回过程是一致的，不同的只是 HTTP 携带的参数。
- 当开启 HTTP gzip 功能模块，并且接收到客户端发来的 HTTP 请求时，客户端会通过

HTTP-headers 中的 Accept-Encoding 属性告诉服务器“我支持 gzip 解压，解压格式（算法）为：deflate, sdch”。

- 注意，并不是 HTTP 说自己支持解压，Nginx 返回响应数据时就一定会压缩。这还要看本次 Nginx 返回数据的格式是什么，如果返回数据的原始数据格式和设置的 gzip\_types 相符合，这时 Nginx 才会进行压缩。
- 接下来根据 HTTP 协议要求，Nginx 处理完成后首先向客户端返回响应头（response headers），如果数据被压缩了，则会在 Content-Encoding 属性中标识 gzip 关键字，表示接下来返回的响应正文（response content）是经过压缩的，并且在 Content-Type 属性中标识数据的原始格式。
- 最后，Nginx 会向客户端返回经过压缩的 response content 信息，客户端再进行解压操作。这里注意一下，在客户端发送的 headers 里面，有一个“deflate, sdch”。这是两种压缩算法，如果读者感兴趣，可以查阅相关资料（了解哈夫曼压缩算法对扩展自己的架构思路很有帮助）。

### 3.5.2 rewrite 模块

本小节内容假定读者已经了解正则表达式的知识。如果你不清楚正则表达式，则请首先查阅相关资料。正则表达式本身不在我们讨论的范围内。

Nginx 的强大在于其对 URL 请求的重写（重定位），这就是 rewrite 功能模块的作用。Nginx 的 rewrite 功能模块依赖于 PCRE Lib，所以请一定在 Nginx 编译安装时，安装 PCRElib。本小节会先讲解 rewrite 的关键语法，然后出示几个示例，并基于示例进行讲解。先来说一下 Nginx 中几个关键的语法。

#### 1. 正在表达式匹配

- ~: 区分大小写进行正则表达式匹配
- ~\*: 不区分大小写进行正则表达式匹配
- !~: 区分大小写进行正则表达式不匹配
- !~\*: 不区分大小写进行正则表达式不匹配

举例说明：

```
示例 1: location ~* \.(jpg|gif|png|ioc|jpeg)$
```

location 是 Nginx 中的关键字，代表当前的 URL 请求值。

以上表达式表示对 URL 进行不区分大小写的匹配，一旦 URL 以 jpg、gifpng、ioc、jpeg 结尾时，匹配成功。



示例 2: `$var1 ~ ^(\d+)$`

`var1` 是 Nginx 中使用 `set` 关键字定义的变量，以上语句代表 `var1` 和一个整数进行匹配。

## 2. Nginx 中的全局变量

从上面的两个实例中，我们已经发现 Nginx 支持变量。不仅如此，Nginx 还内置了一些全局变量，这里列举一些比较重要的全局变量。

- `$content_length`: 获取 request 中 header 部分的“Content\_Length”值。
- `$content_type`: 获取 request 中 header 部分的“Content\_type”值。
- `$request_method`: 请求方式，常用的有两种请求方式：POST、GET。
- `$remote_addr`: 发送请求的客户端 IP。
- `$remote_port`: 发送请求的客户端端口。
- `$request_uri`: 含有参数的完整的初始 URI。
- `$server_addr`: request 到达的 server 的 IP。
- `$server_port`: 请求到达的服务器的端口号。

## 3. rewrite 语法

`rewrite regex replacement flag`

#`regex`: 表示当前匹配的正则表达式。只有 `$url` 大小写匹配 `regex` 正则表达式，这个 `$url` 才会被 `rewrite` 进行重定向

#`replacement`: 重定向目标规则。这个目标规则支持动态变量绑定，这个问题下面会用示例来讲述  
#`flag`: 重定向规则

## 4. rewrite 中的 Flag 关键字

- `redirect`: 通知客户端重定向到 `rewrtie` 后面的地址。
- `permanent`: 通知客户端永久重定向到 `rewrtie` 后面的地址。
- `last`: 将 `rewrite` 后的地址重新在 `server` 标签中执行。
- `break`: 将 `rewrite` 后的地址重新在当前的 `location` 标签中执行。

实际上，对于客户端来说，其效果是一样的，都是由客户端重新发起 HTTP 请求，请求地址重新定位到 `replacement` 规则的 URL 地址；这里关键要讲解最常用的 `last` 和 `break` 两个关键字。

所有的 `rewrite` 语句都是要在 `server` 中的 `location` 中书写，如下：

```
server {
    .....
    location ... {
        if(...) {
            rewrite regex replacement flag;
        }
    }
}
```

```

        rewrite regex replacement flag;
    }
}

```

那么，`break` 关键字说明重写的 `replacement` 地址在当前 `location` 的区域马上执行。

`last` 关键字说明重写的 `replacement` 地址在当前 `server` 所有的 `location` 中重新再做匹配。

下面我们结合 `rewrite` 关键字和 `rewrite flag` 关键字给出典型的示例进行讲解。

#### 示例 1:

```

location ~* ^/(.+)/(.+)\.(jpg|gif|png|jpeg)$ {
    rewrite ^/orderinfo/(.+)\.(jpg|gif|png|jpeg)$ /img/$1.$2 break;
    root /cephclient;
}

```

`location` 在不进行大小写区分的情况下利用正则表达式对 `$url` 进行匹配。当匹配成功后进行 `rewrite` 重定位。

`rewrite` 重写 URL 的规则是：`regex` 表达式第一个括号中的内容对应 `$1`，`regex` 表达式第二个括号中的内容对应 `$2`，以此类推。

这样重定位的意义就很明确了：将任何目录下的文件名重定位到 `img` 目录下的对应文件名，并且马上在这个 `location` 中（注意是 `Nginx`，而不是客户端）执行这个重写后的 URL 定位。

#### 示例 2:

```

server {
    .....
    location ~* ^/orderinfo/(.+)\.(jpg|gif|png|jpeg)$ {
        rewrite ^/orderinfo/(.+)\.(.+)$ /img/$1.$2 last;
    }
    location / {
        root /cephclient;
    }
}

```

在 `server` 中，有两个 `location` 位置，当 URL 需要访问 `orderinfo` 目录下的某一个图片时，`rewrite` 将重写这个 URL，并且重新带入这个 URL 到 `server` 执行，这样“`location /`”这个 `location` 就会执行了，并找到图片存储的目录。

### 3.5.3 健康检查模块

本小节将介绍一个用于 `Nginx` 对后端 `UpStream` 集群节点健康状态检查的第三方模块：`nginx_upstream_check_module`（[https://github.com/yaoweibin/nginx\\_upstream\\_check\\_module](https://github.com/yaoweibin/nginx_upstream_check_module)）。



有资料介绍这个模块是淘宝团队开发的，但是笔者在 GitHub 上试图求证时并没有找到直接证据。

这里需要说明的是，目前有很多 Nginx 模块可以实现 Nginx 对后端集群节点的健康监测，不止 `nginx_upstream_check_module`。Nginx 官方有一个模块 `healthcheck_nginx_upstreams` 也可以实现对后端节点的健康监测（[https://github.com/cep21/healthcheck\\_nginx\\_upstreams](https://github.com/cep21/healthcheck_nginx_upstreams) 有详细的安装和使用介绍）。

我们回到对 `nginx_upstream_check_module` 的讲解，要使用这个第三方模块首先需要进行下载，然后通过 `patch` 命令将补丁打入原有的 Nginx 源码中，并且重新进行编译安装。下面来重点讲解一下这个模块的安装和使用。

下载 `nginx_upstream_check_module` 模块：

```
wget https://codeload.github.com/yaoweibin/nginx_upstream_check_module/zip/master
```

你也可以直接到 GitHub 上进行下载，还可以在 Linux 系统上使用 `git` 命令进行下载。

解压安装，并将该模块打入 Nginx 源码：

```
# unzip ./nginx_upstream_check_module-master.zip
```

注意是将补丁打入 Nginx 源码，不是 Nginx 的安装路径：

```
# cd ./nginx-1.6.2
# patch -p1 < ../nginx_upstream_check_module-master/check_1.5.12+.patch
```

如果补丁安装成功，你将看到以下的提示信息：

```
patching file src/http/modules/nginx_http_upstream_ip_hash_module.c
patching file src/http/modules/nginx_http_upstream_least_conn_module.c
patching file src/http/nginx_http_upstream_round_robin.c
patching file src/http/nginx_http_upstream_round_robin.h
```

这里请注意，在 `nginx_upstream_check_module` 官网的安装说明中，有一个打补丁的注意事项：

```
If you use nginx-1.2.1 or nginx-1.3.0, the nginx upstream round robin
module changed greatly. You should use the patch named
'check_1.2.1.patch'.
If you use nginx-1.2.2+ or nginx-1.3.1+, It added the upstream
least_conn module. You should use the patch named 'check_1.2.2+.patch'.
If you use nginx-1.2.6+ or nginx-1.3.9+, It adjusted the round robin
module. You should use the patch named 'check_1.2.6+.patch'.
If you use nginx-1.5.12+, You should use the patch named
'check_1.5.12+.patch'.
```

```
If you use nginx-1.7.2+, You should use the patch named  
'check_1.7.2+.patch'.
```

# 这里我们的 Nginx 的版本是 1.6.2, 那么就应该打入 check\_1.5.12+.patch 这个补丁

重新编译安装 Nginx:

注意重新编译 Nginx, 要使用 add-module 参数将这个第三方模块安装进去:

```
# ./configure --prefix=/usr/nginx-1.6.2/ --add-module=../nginx_upstream_  
check_module-master/  
# make && make install
```

通过以上步骤, 第三方的 nginx\_upstream\_check\_module 模块就在 Nginx 中准备好了。接下来讲解如何使用这个模块。首先看一下 upstream 的配置信息:

```
upstream cluster {  
    # simple round-robin  
    server 192.168.0.1:80;  
    server 192.168.0.2:80;  
    check interval=5000 rise=1 fall=3 timeout=4000;  
    #check interval=3000 rise=2 fall=5 timeout=1000 type=ssl_hello;  
    #check interval=3000 rise=2 fall=5 timeout=1000 type=http;  
    #check_http_send "HEAD / HTTP/1.0\r\n\r\n";  
    #check_http_expect_alive http_2xx http_3xx;  
}
```

上面的代码中, check 部分就是调用 nginx\_upstream\_check\_module 模块的语法:

```
check interval=milliseconds [fall=count] [rise=count]  
[timeout=milliseconds] [default_down=true|false]  
[type=tcp|http|ssl_hello|mysql|ajp|fastcgi]
```

- interval: 必要参数, 检查请求的间隔时间。
- fall: 当检查失败次数超过了 fall, 这个服务节点就变成 down 状态。
- rise: 当检查成功的次数超过了 rise, 这个服务节点又会变成 up 状态。
- timeout: 请求超时时间, 超过这个时间后, 这次检查就算失败。
- default\_down: 后端服务器的初始状态。默认情况下, 检查功能在 Nginx 启动的时候将会把所有后端节点的状态置为 down, 检查成功后, 再置为 up。
- type: 这是检查通信的协议类型, 默认为 HTTP。以上类型是检查功能所支持的所有协议类型。

```
check_http_send http_packet  
http_packet 的默认格式为: "GET / HTTP/1.0\r\n\r\n"
```

`check_http_send` 设置，这个设置描述了检查模块在每次检查时，向后端节点发送什么样的信息。

```
check_http_expect_alive [ http_2xx | http_3xx | http_4xx | http_5xx ]
```

这些状态代码表示服务器的 HTTP 响应是没问题的，后端节点是可用的。默认情况的设置是：`http_2xx | http_3xx`。当你根据自己的配置要求完成检查模块的配置后，请首先使用 `nginx-t` 命令监测配置文件是否可用，然后再用 `nginx -s reload` 命令重启 Nginx。

### 3.5.4 图片动态缩略模块

在实际业务开发过程中，我们常遇到这样的问题：一张 1000×1000 像素的 PNG 图片上传后需要按照不同的等比例像素显示在不同平台上。例如在网页上将这张原始图片按照 600×600 像素显示出来，在手机上按照 400×400 像素显示出来，在平板设备的图片列表页面上按照 100×100 像素进行显示。并且由于外部各种因素的变化，在不同设备上显示同一张图片的大小也在不停地变化。

通常读者会怎样处理这样的业务功能呢？最直接最能够立即想到，也是处理方式最简单的办法，就是将图片按照原始大小传输到客户端，再由客户端的 Html5 代码、Android 代码、Object-C 代码等客户端编程语言进行图片处理。但是这样做的缺点也是很明显的：1000×1000 像素的 PNG 图片大小普遍在 100KB~200KB 之间，这会导致多张图片传输时占用巨大的服务器带宽资源。如果有 100 个用户同时请求图片资源，那么瞬时带宽将达到 100Mbps~200Mbps。

当然，还有另外的处理方式可以显著节约带宽资源：当这个 1000×1000 像素的原始图片上传时，服务器端除了保存原图，还同时按照客户端当前的图片尺寸要求，分别生成 600×600 像素、400×400 像素、100×100 像素的缩略图一同保存到服务器端。当客户端需要使用同一张图片的不同像素文件时，调用不同的保存路径就行了。一张 400×400 像素的 PNG 图片大小普遍为 20KB~40KB，显然这比传输原始图片本身减轻了不少的网络带宽压力。当然这样做的问题还是很明显的：

- 如果要在图片文件完成上传时就生成固定大小的缩略图，那么这些缩略图的尺寸从开始就最好是固定的。很难想象当客户端团队要求为每张商品原图增加一张 300×300 像素的缩略图时，服务端开发人员能在很短的时间内完成目前已存放在服务端的 20 000 000 张商品原图的新尺寸缩略图的生成动作。
- 磁盘空间也是一个问题，正式系统的图片文件一般存放在分布式文件系统上，例如 MFS 分布式文件系统、Ceph 分布式文件系统，又或者是类似淘宝的 TFS 分布式文件系统，当然采用块存储方案的图片服务器也是有的（例如 EMC 提供的大型盘阵系统）。它们的共同特点都是可以提供海量的文件存储空间（例如 100TB）。但是技术团队的

要求一定都是利用有限的空间尽可能地多存储数据。所以如果使用类似的方法, 200KB 的原始文件就可能需要 400KB 甚至更多的磁盘空间进行存储, 这样显然是不太符合要求的。

Nginx 的动态缩略图模块为本小节讨论的功能需求提供了一个比以上两种解决办法都要好的方案: 它可以将指定路径下的图片按照技术团队的要求动态生成等比例缩略图片, 并且不耗费额外的磁盘存储空间, 通过正则表达式和 `rewrite` 模块的支持它可以随心所欲调整缩略图的大小。

如果你需要使用 Nginx 的动态缩略图模块, 就需要重新编译安装 Nginx, 并在安装时指定。

```
# yum install gd-devel
# ./configure [这里该加什么参数加什么参数,
# 例如 prefix] --with-http_image_filter_module
# make && make install
```

其他安装过程和上文介绍过的没有任何差别。现在我们来看看 `image_filter` 模块的主要语法:

- `image_filter test`: 测试当前原始图片是否是 JPEG、GIF 或 PNG 图片, 如果测试过程出现错误则返回 415。
- `image_filter rotate 90 | 180 | 270`: 这个语法可以让 `image_filter` 模块按照 90°、180°、270° 这 3 个角度旋转图片。
- `image_filter size`: 以 JSON 格式输出图片宽度、高度、类型。
- `image_filter resize [width] [height]`: 在保持图片比例的前提下, 根据给定的长宽生成缩略图。
- `image_filter crop [width] [height]`: 以其最大边缩放图片后截取多余的部分。
- `image_filter jpeg_quality`: 设置 JPEG 图片的压缩质量比例, 例如设置为 75, 则表示 JPEG 图片的压缩质量比为 75%。
- `image_filter_buffer`: 单张原始图片最大大小, 默认为 1MB; 如果原始图片超过这个值, 则 `image_filter` 模块会报错。按照服务器端存储的图片大小, 最好设置一个兼容性较好的值。

以下是一段 `image_filter` 模块在 Nginx 配置文件中的设置信息:

```
.....
# 不区分大小写进行正则表达式匹配
# 只要满足以下表达式的请求路径, 则进行图片动态缩略图的生成
location ~* /\.+(jpg|gif|png|ioc|jpeg)_(\d+)_(\d+)$ {
    # 将以上正则表达式第 3 个区域设置的数值传入变量 “w”
```



```

set $w $3;
# 将以上正则表达式第 4 个区域设置的数值传入变量 “h”
set $h $4;
# 将 rewrite 后的地址重新在当前的 location 标签执行, 以便找到原始图片
rewrite /(.)\.(jpg|gif|png|ioc|jpeg)_(\d+)_(\d+)$ /$1.$2 break;
# 按照宽度 w 和高度 h, 将原始图片进行等比例缩放
image_filter resize $w $h;
# 设置单张原始图片的最大值为 10MB
image_filter_buffer 10M;
# 设置图片所在的根目录
root /home/nodejs/BDK/public/upload/;
}

```

Nginx 配置文件的其他部分不会受到图片模块的影响, 所以之前是怎么设置的现在还是怎么设置。这里为了节约篇幅就不再重复列出了。以下是不同 HTTP 请求路径显示的不同图片比例效果。

- [http://10.5.1.249:2999/proposal/QJ0LyJcPU9kxa.png\\_800\\_800](http://10.5.1.249:2999/proposal/QJ0LyJcPU9kxa.png_800_800) (图 3-12)

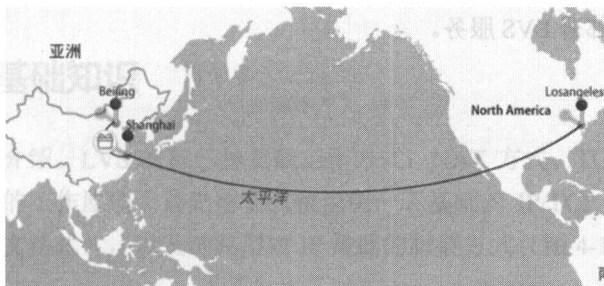


图 3-12 800×800 像素图片

- [http://10.5.1.249:2999/proposal/QJ0LyJcPU9kxa.png\\_400\\_400](http://10.5.1.249:2999/proposal/QJ0LyJcPU9kxa.png_400_400) (图 3-13)

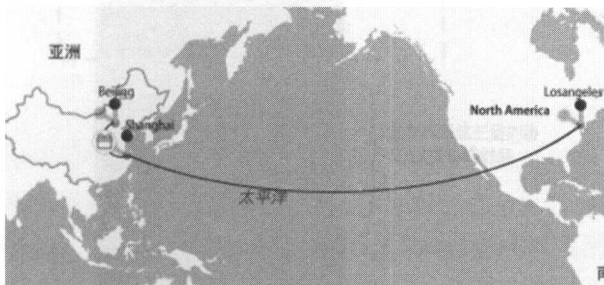


图 3-13 400×400 像素图片

- [http://10.5.1.249:2999/proposal/QJ0LyJcPU9kxa.png\\_200\\_200](http://10.5.1.249:2999/proposal/QJ0LyJcPU9kxa.png_200_200) (图 3-14)

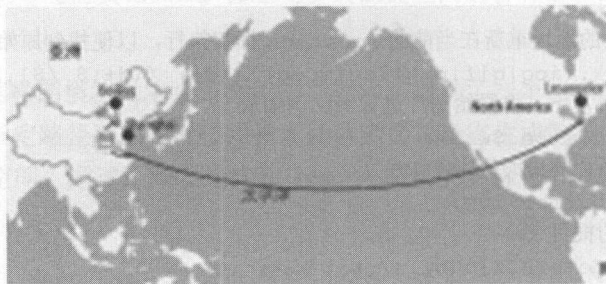


图 3-14 200×200 像素图片

读者还可以通过调整 `location` 部分的正则表达式格式, 调整 HTTP 请求路径的格式。需要注意的是, **Nginx** 动态缩略图模块的工作非常消耗 **CPU** 和 **内存资源**——这倒不是因为这个模块的代码性能有问题, 而是因为处理位图本来就消耗 **CPU** 和 **内存资源**, 在本书第四部分场景实战中还会提到这个问题。所以如果你确定要用这个方案, 那么建议为物理服务器配备性能更好的 **CPU** 和更大的内存。另外在正式系统部署之初就最好准备多个 **Nginx** 服务节点, 并在这些 **Nginx** 服务节点前部署 **LVS** 服务。

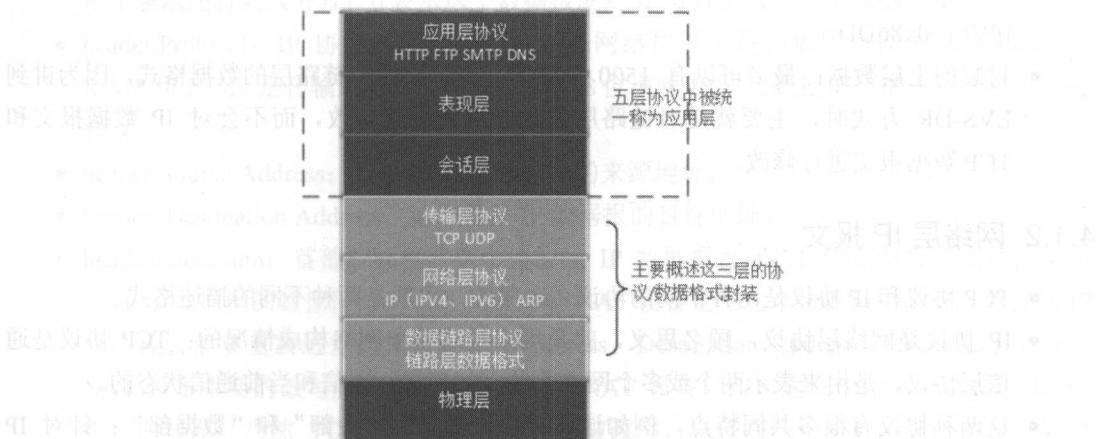
# 第 4 章

## LVS 技术

LVS 是章文嵩博士发起的开源项目，全称是 Linux Virtual Server，从字面上的理解就是 Linux 虚拟服务器。实际上 LVS 在生产环境下的主要实践是建立一个存在于网络层或链路层上的负载均衡管理器。本章内容主要向读者介绍 LVS 的基本工作原理和使用实例。

### 4.1 网络协议基础知识

根据官方文档的介绍，LVS 支持三种负载工作方式：NAT 方式、TUN 方式和 DR 方式。为了说明这三种方式的工作原理，首先需要大致说明一下基础的 IP/TCP 报文（IP 报文和 TCP 报文是两种不同的报文格式），以及链路层对 IP 数据的封装方式（图 4-1）。



4-1 七层/五层网络协议



LVS 体系之所以高效是因为其直接对链路层报文、IP 报文、TCP 报文进行了修改或封装。为了保证读者不会对网络协议的理解发生偏差，我们只介绍其中需要用到的属性和含义，如果读者对网络的核心原理有兴趣，那么可以读一读《TCP/IP 详解，卷 1：协议》这本书。

4.1.1 链路层报文

链路层的数据格式有一个共同特点，都包括目标 MAC 地址和源 MAC 地址。如图 4-2 所示主要说明了最常用的 Ethernet 帧（以太帧）的报文格式。

目标MAC地址 00:00:00:00:00:00	源MAC地址 FF:FF:FF:FF:FF:FF	上层协议类型	封装的上层数据	校验串
------------------------------	-----------------------------	--------	---------	-----

图 4-2 链路层报文

- 目标 MAC 地址/源 MAC 地址：00:00:00:00:00:00——FF:FF:FF:FF:FF:FF 这个范围是全球 MAC 地址的可用范围。一张物理网卡肯定有一个唯一的 MAC 地址。实际上网络层常用的 IP 协议，就是基于 MAC 地址的。一个子网范围内某个 IP 对应的 MAC 地址是通过 ARP 查询协议从 NAT 设备（可能是路由器、交换机或者网络代理设备）上查询得到的。
- 上层协议类型：链路层的报文是为了承载网络层的协议而存在的，所以链路层的数据格式中需要有一个属性说明这个链路层所承载的上层协议是什么类型。例如：  
IPV4: 0x0800  
ARP: 0x0806  
PPPoE: 0x8864  
IPV6: 0x86DD
- 封装的上层数据：最多可以有 1500 个字节。请记住这个链路层的数据格式，因为讲到 LVS-DR 方式时，主要就是对链路层的数据格式进行修改，而不会对 IP 数据报文和 TCP 数据报文进行修改。

4.1.2 网络层 IP 报文

- TCP 协议和 IP 协议是两种不同的协议。对应的，也就是两种不同的描述格式。
- IP 协议是网络层协议，顾名思义，就是用来描述整个网络构成情况的；TCP 协议是通信层协议，是用来表示两个或多个网络上的点如何进行通信和当前通信状态的。
- 这两种协议有很多共同特点，例如这两种协议都分为“头部”和“数据部”；针对 IP 协议来说，TCP 协议的描述就存放在其“数据部”中。

图 4-3 来源于网路，其中有几个重要的后面要使用到的属性，下面先进行说明。

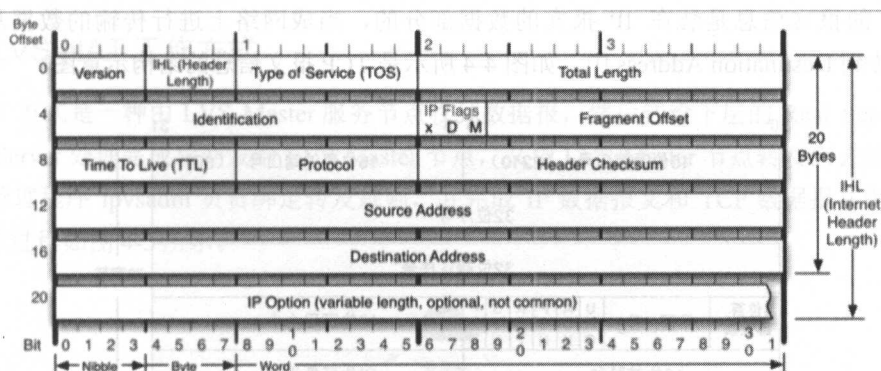


图 4-3 网络层 IP 报文

- **header.version:** IP 协议的版本号，既是 IPV4 也是 IPV6。4 表示 IPV4 版本，6 表示 IPV6。你要问我什么是 IPV4，什么又是 IPV6；192.168.220.141，就是 IPV4 的格式；FE80:0000:0000:AAAA:0000:00C2:0002，就是 IPV6 的格式。
- **header.Total Length:** 总长度，这个总长度是 IP 头和 IP 数据两个区域的总长度。主要还是用于生成头验证码和为了操作系统处理方便。
- **header.IP Flags:** 这个位置有三位 000，但实际只有后两位才有值 010，这个就是“D”位=1，这个时候表示由于要传输的整个数据不大，所以这个 IP 数据报的数据部分已经描述了整个数据描述，不需要进行 IP 数据报的分片。“D”位=0，表示要传输的整个数据比较大，所以 IP 数据报进行了拆分，这时就要用到最后一位了：最后一位“M”中 1 表示还有后续分片；0 表示这个数据报就是 IP 分片的最后一个数据片了。
- **header.Protocol:** IP 协议是网络层协议，在网络层以上是传输层协议。TCP、UDP、ICMP 和 IGMP 是传输层协议。这个位置的 8 位说明 IP 协议数据部分携带的是哪种上层协议。
- **header.Source Address:** 这个就是 IP 数据报的来源地址。
- **header.Destination Address:** 这个就是 IP 数据报的目标地址。
- **header.checksum:** 首部校验值。这个值校验 IP 数据报首部的传输完整性（注意校验不包括 IP 数据报的数据部分）。这就意味着 NAT 设备重写这个数据报的来源或者目标 IP 后，校验值要重新进行计算。Source Address、Destination Address、Checksum 是各种 NAT 设备主要的改写属性。而且很多时候 NAT 设备只改写这三个值就可以实现 IP 数据报的转发（当然 TCP 报文中的端口也会被改写，以便在端口映射的情况下进行端口转换）。

4.1.3 传输层 TCP 报文

TCP 的报文信息是装在 IP 报文的数据部分的，当成网络上进行传输的数据从 Srouce Address 传到 Destination Address 中。如图 4-4 所示是 TCP 报文信息的结构示意图。

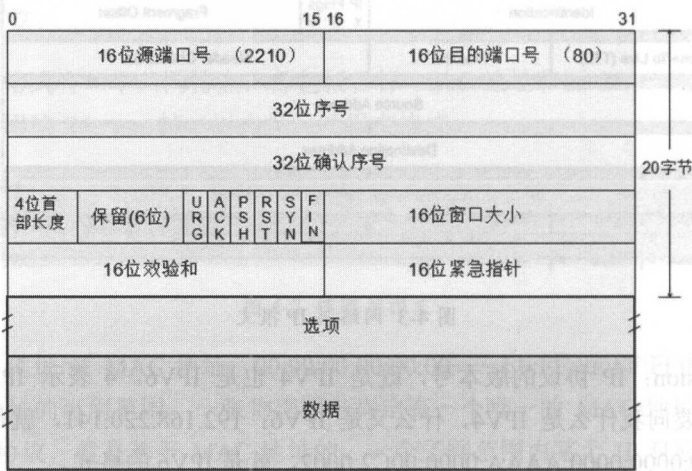


图 4-4 TCP 报文结构

- 头.源端口号：TCP 信息来源的端口号。
- 头.目的端口号：TCP 信息数据的目标端口。
- 头.状态位（URG/ACK/PSH/RST/SYN/FIN）： TCP 的三次握手和连接中断都要用到三个标记 ACK、SYN、FIN。需要注意 SYN SEQ 和 ACK SEQ 就是 TCP 数据报的确认信号。另外解释一下 PSH 和 RST 两个状态标记：应用层的 TCP 数据报有一个缓存区，也就是说多个正确的 TCP 数据报会首先放到这个缓存区，达到一定条件后，再推送给上层的应用层协议，例如 HTTP 协议。PSH 为 1 的时候，表示不需要再等到后续的 TCP 数据报文了，直接将目前接收方缓存中的 TCP 数据报进行数据段组合后推送给上层协议，并且清空缓存区；RST 表示复位，你可以理解成放弃当前缓存区的所有未发送给上层协议的 TCP 数据报文，一般这种情况都是 TCP 报文传输出现了问题。
- 头.TCP 校验和：TCP 报文的校验和比起 IP 头校验要稍微复杂点。TCP 校验的输入包括三部分：TCP 伪首部、TCP 首部长度和 TCP 数据部长度。TCP 伪首部是一个虚拟概念，它包括承载 TCP 数据报文的 IP 报文的一部分，和 TCP 首部的一部分数据（源 IP、目标 IP、IP 报文中的协议，以及 TCP 报文的报文头长度和 TCP 报文的数据长度）。
- 一旦 IP 报文中的源 IP 和目标 IP 发生改变，TCP 报文校验信息就会改变。

## 4.2 LVS 的三种工作方式

### 4.2.1 LVS-NAT 工作方式

NAT 方式是一种由 LVS Master 服务节点收到数据报，然后转给下层的 Real Server 节点，当 Real Server 处理完成后回发给 LVS Master 节点，又由 LVS Master 节点转发出去的工作方式。LVS 的管理程序 ipvsadm 负责绑定转发规则，并完成 IP 数据报文和 TCP 数据报文中属性的重写。工作过程如图 4-5 所示。

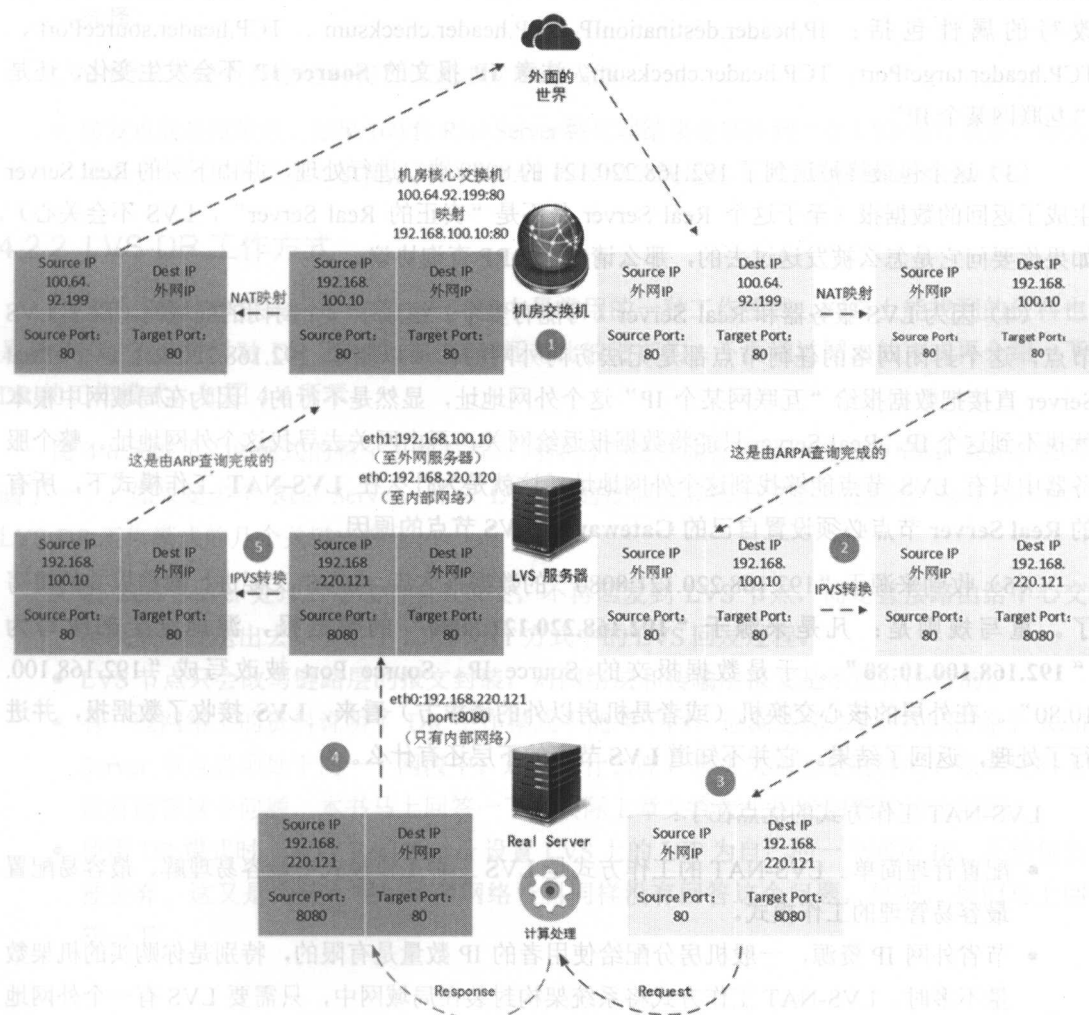


图 4-5 NAT 工作模式

(1) 在正式的机房环境中，一般有两种方式为一个机器分配外网地址：在核心交换机上直接绑定外网地址到主机网卡的，这样使用 `ifconfig` 命令看到的 IP 地址为外网地址；或者在核心交换机上使用映射规则，将一个外网地址映射到内网地址，这样使用 `ifconfig` 命令看到的 IP 地址为内网地址。图 4-5 中我们采用的是后一种映射规则。如果使用前一种外网 IP 的分配规则，也不会影响 LVS-NAT 的工作方式，因为这个 IP 被限制在 LVS-NAT 工作以外。只不过 `eth1` 的 IP 从 192.168.100.10 换成了 100.64.92.199。

(2) 我们用中文描述一下转换规则：凡是发送到“192.168.100.10:80”的数据报，目标地址全部改写为“192.168.220.121:8080”，所以来自于 100.64.92.199:80 的报文被改写了。被改写的属性包括：IP.header.destinationIP、IP.header.checksum、TCP.header.sourcePort、TCP.header.targetPort、TCP.header.checksum。注意 IP 报文的 Source IP 不会发生变化，还是“互联网某个 IP”。

(3) 这个包最终被送到了 192.168.220.121 的 8080 端口进行处理，并由下层的 Real Server 生成了返回的数据报（至于这个 Real Server 是不是“真正的 Real Server”，LVS 不会关心）。如果你要问它是怎么被发送过去的，那么请参考 ARP 查询协议。

(4) 因为 LVS 服务器和 Real Server（可能有多个）组成了一个封闭的局域网。除了 LVS 节点，这个封闭网络的任何节点都是无法访问外网的。所以要求 192.168.220.121 这个 Real Server 直接把数据报给“互联网某个 IP”这个外网地址，显然是不行的，因为在局域网中根本就找不到这个 IP。Real Server 只能将数据报返给网关，再由网关去寻找这个外网地址。整个服务器中只有 LVS 节点能够找到这个外网地址，这就是为什么在 LVS-NAT 工作模式下，所有的 Real Server 节点必须设置自己的 Gateway 为 LVS 节点的原因。

(5) 收到来源于“192.168.220.121:8080”的数据报文后，IPVS 又要进行数据报文的重写了。重写规则是：凡是来源于“192.168.220.121:8080”的数据报，源地址全部改写为“192.168.100.10:80”。于是数据报文的 Source IP、Source Port 被改写成“192.168.100.10:80”。在外层的核心交换机（或者是机房以外的请求方）看来，LVS 接收了数据报，并进行了处理，返回了结果。它并不知道 LVS 节点的下层还有什么。

LVS-NAT 工作方式的优点在于：

- 配置管理简单。LVS-NAT 的工作方式是 LVS 三种工作模式中最容易理解、最容易配置、最容易管理的工作模式。
- 节省外网 IP 资源，一般机房分配给使用者的 IP 数量是有限的，特别是你购买的机架数量不多时。LVS-NAT 工作方式将系统架构封装在局域网中，只需要 LVS 有一个外网地址或外网地址映射就可以实现访问了。



- 系统架构相对封闭。在内网环境下我们对防火墙的设置要求不会很高，也相对容易进行物理服务器的运维。你可以设置来源于外网的请求需要进行防火墙过滤，而对内网请求开放访问。
- 另外改写后转给 Real Server 的数据报文，Real Server 并不会关心它的真实性，只要 TCP 校验和 IP 校验都能通过，Real Server 就可以进行处理。所以，LVS-NAT 工作模式下 Real Server 可以是任何操作系统，只要它支持 TCP/IP 协议即可。
- 当然作为 Linux 系统的忠实拥护者，笔者并不建议使用 Windows 服务器。但如果你的 Real Server 是 .Net 系统，又有业务场景需要用到 LVS，那么 LVS-NAT 可能是一个不错的选择。

LVS-NAT 的缺点是由这种转发模式本身所造成的：

- 转发点就是瓶颈点。如果 100 台 Real Server 将处理结果全部转到一个 LVS 进行发送，那么将是一个怎样的场景。事实上，LVS-NAT 的极限负载是达不到 100 台 Real Server 的。

## 4.2.2 LVS-DR 工作方式

LVS 的 DR 工作模式，是目前生产环境中最常用的一种工作模式，网络上能找到的资料也是最多的。部分资料对 DR 工作模式的讲解还是比较透彻的。下面通过图文的方式再介绍一下 DR 的工作模式，如图 4-6 所示。

图 4-6 反映了 DR 模式的整个工作过程，同样为了便于读者理解，这里的 Real Server 也只画了一个。如果是多个 Real Server，那么 LVS 会通过调度算法来决定发往哪台 Real Server。LVS-DR 工作模式的几个关键点在于：

- 被 Real Server 处理后形成的响应报文，不再回发到 LVS 节点，而是直接路由给中心交换机然后发送出去。省去了 LVS-NAT 方式中的 LVS 回发过程。
- LVS 节点只会改写链路层的报文封装，对网络层和传输层报文是不进行改写的。
- 有一些网络上的资料说明了 DR 工作模式不能跨子网，也就是说 LVS 节点和各个 Real Server 节点必须处于同一个网段中。这是为什么呢？事实又真的是这样吗？很多资料上没有回答这个问题，本书马上回答一下（实际上章文嵩先生已经回答过这个问题）。
- 使用 DR 模式时，需要 Real Server 设置 LVS 上的 VIP 为自己的一个回环 IP，不然包会被丢弃。这又是为什么呢？很多网络资料同样没有回答这个问题，好吧，我们马上回答一下。

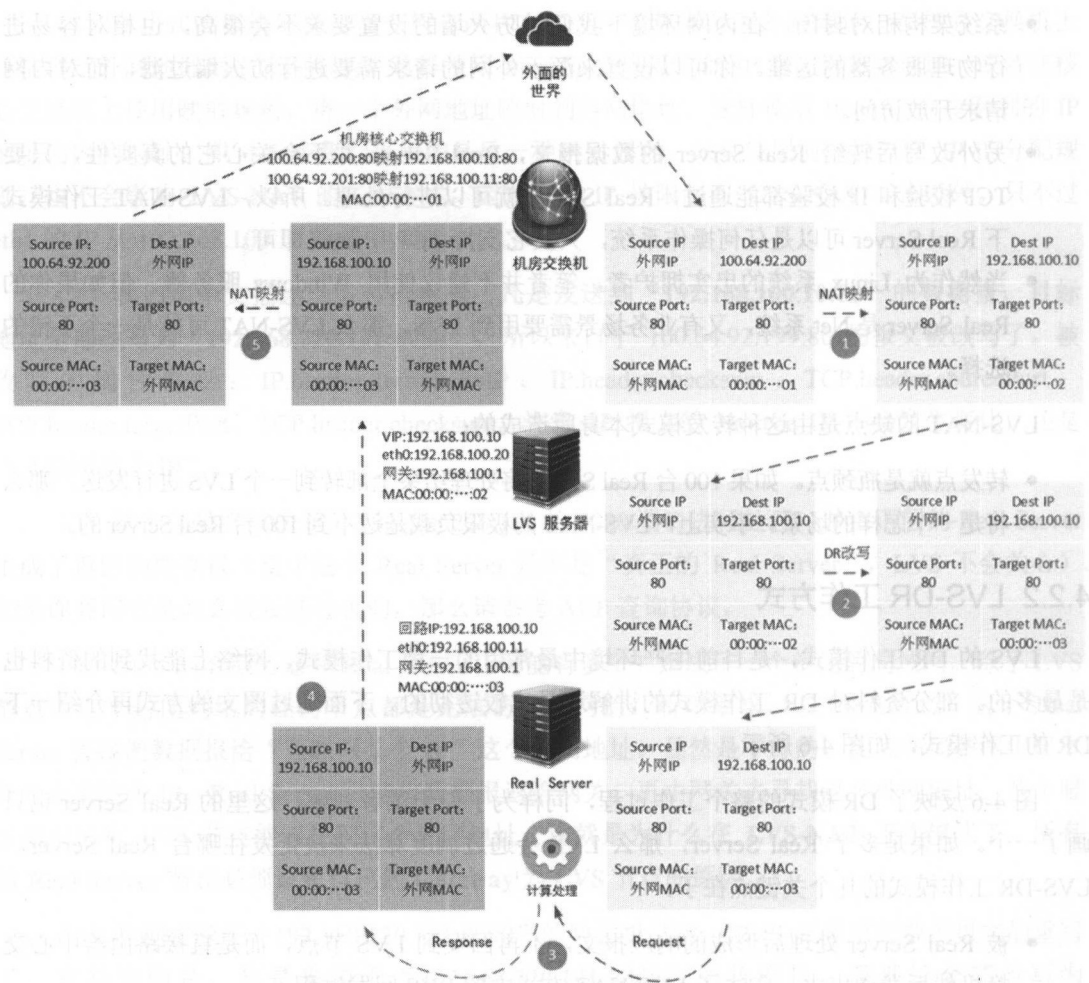


图 4-6 DR 工作模式

图 4-6 中 DR 模式的工作过程为：

- (1) 中心交换机采取的同样是 IP 映射方式，但是与 LVS-NAT 方式不一样，Real Server 在机房的中心交换机上也需要绑定一个外网映射。这样可以保证 Real Server 回发的响应报文能够被发送到外网。
- (2) LVS 节点接收到请求报文后，会改写报文的数据链路层格式。将 Target Mac 改写成 Real Server 的 Mac，但是网络层和传输层报文不会改写，然后重新回发给交换机。这里就涉及一个问题，现在 Target Mac 和 Destination IP 的对应关系是错误的，这个数据报文到了交换机后，由于这种错位的关系，是不能进行三层交换的，只能进行二层交换（一旦进行 IP 交换，



数据报文的验证就会出错，最终被丢弃）。所以 LVS-DR 方式要求 Real Server 和 LVS 节点必须在同一个局域网内。或者更确切的说法是：LVS 节点需要找到一个二层链路，将改写了 Mac 地址的报文发送给 Real Server，而不能进行三层交换的校验。这样来看，实际上 LVS 节点和 Real Server 界面不一定要在同一个子网，用一个独立网卡独立组网，传送报文也是可行的。

（3）通过二层交换，数据被发送到 Real Server 节点。那么 Real Server 节点怎么来判断这个包的正确性？首先当然是传输层 TCP/IP 报文校验没有问题，若 LVS-NAT 没有改写 TCP/IP，那么校验就没有问题（除非报本身就存在问题）；然后是链路层的 Mac 地址能够被识别，这时就是回环 IP 的功劳了。对于 Real Server 节点来说，192.168.100.10 这个 VIP 就是自己的回环 IP，绑定的 Mac 也就是被 LVS 替换后的 Target Mac。那么 Real Server 会认为这个包是在本机运行的某一个应用程序通过回环 IP 发给自己的，所以这个包不能被丢弃，必须处理。

（4）被处理后生成的响应报文，被直接发送给网管。这个就没有太多需要解释的了，只要保证 Real Server 的默认路由设置成到核心交换机的 192.168.100.1 就可以了。另外，需要说明的是，LVS-DR 模式并没有更改原有的 IP 报文和 TCP 报文，所以 LVS-DR 模式本身是不支持端口映射的，实际上在工作实践中，我们一般使用 Nginx 做端口映射，因为其更灵活。

LVS-DR 工作模式的优点在于：

- 解决了 LVS-NAT 工作模式中的转发瓶颈问题，能够支撑规模更大的负载均衡场景。
- 比较耗费网外 IP 资源，机房的外网 IP 资源都是有限的，如果在正式生产环境中确实存在这个问题，那么可以采用 LVS-NAT 和 LVS-DR 混合使用的方式来缓解。

LVS-DR 当然也有缺点：

- 配置工作较 LVS-NAT 方式稍微麻烦一点，至少需要了解 LVS-DR 模式的基本工作方式才能更好地指导自己解决 LVS-DR 模式的配置和运行过程中遇到的问题。
- 由于 LVS-DR 模式的报文改写规则，导致 LVS 节点和 Real Server 节点必须在一个网段，因为二层交换是没法跨子网的。但是这个问题是针对大多数系统架构方案来说的，实际上并没有本质限制。

### 4.2.3 LVS-TUN 工作方式

很多网络上的资料都说 DR 与 TUN 的工作方式类似，或是直接讲解 DR 模式和 TUN 模式的安装配置方式，然后总结出两种模式是类似的。那为什么有了 DR 模式后还需要 TUN 模式呢？为什么 ipvsadm 针对两种模式的配置参数不一样呢？

实际上，LVS-DR 模式和 LVS-TUN 模式的工作原理完全不一样，工作场景完全不一样。DR 模式基于数据报文重写，TUN 模式基于 IP 隧道，后者是对数据报文的重新封装。下面就

来讲解一下 LVS-TUN 模式的工作原理。

首先要介绍一个概念——IPIP 隧道。将一个完整的 IP 报文封装成另一个新的 IP 报文的数据部分，并通过路由器传送到指定地点。在这个过程中路由器并不在意被封装的原始协议的内容。到达目的地后，由目的地地方依靠自己的计算能力和对 IPIP 隧道协议的支持，打开封装协议，取得原始协议。如图 4-7 所示。

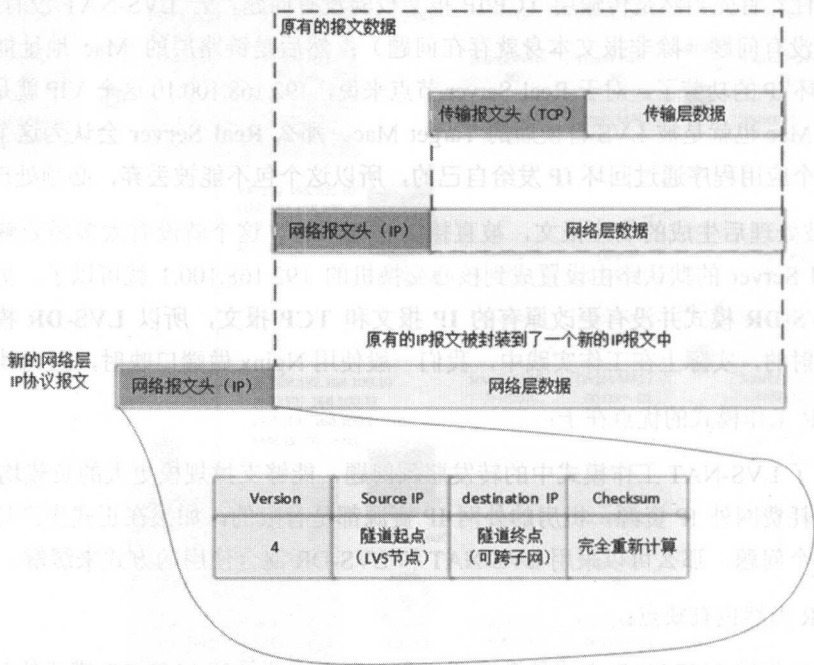


图 4-7 报文封装

可以说隧道协议就是为了解决跨子网传输而准备的，在生产环境中由于业务需要、技术需要或者安全需要，可能使用交换机进行 VLAN 隔离（即形成若干个虚拟的独立的局域网），我们可能需要 LVS 节点在局域网 A，而需要进行负载的多台 MySQL 服务器可能在局域网 B 中。这个时候，就要配置 LVS 的隧道方式。LVS-TUN 模式如图 4-8 所示（注意，目标节点要能够解开隧道协议，好消息是 Linux 支持 IPIP 隧道协议）。

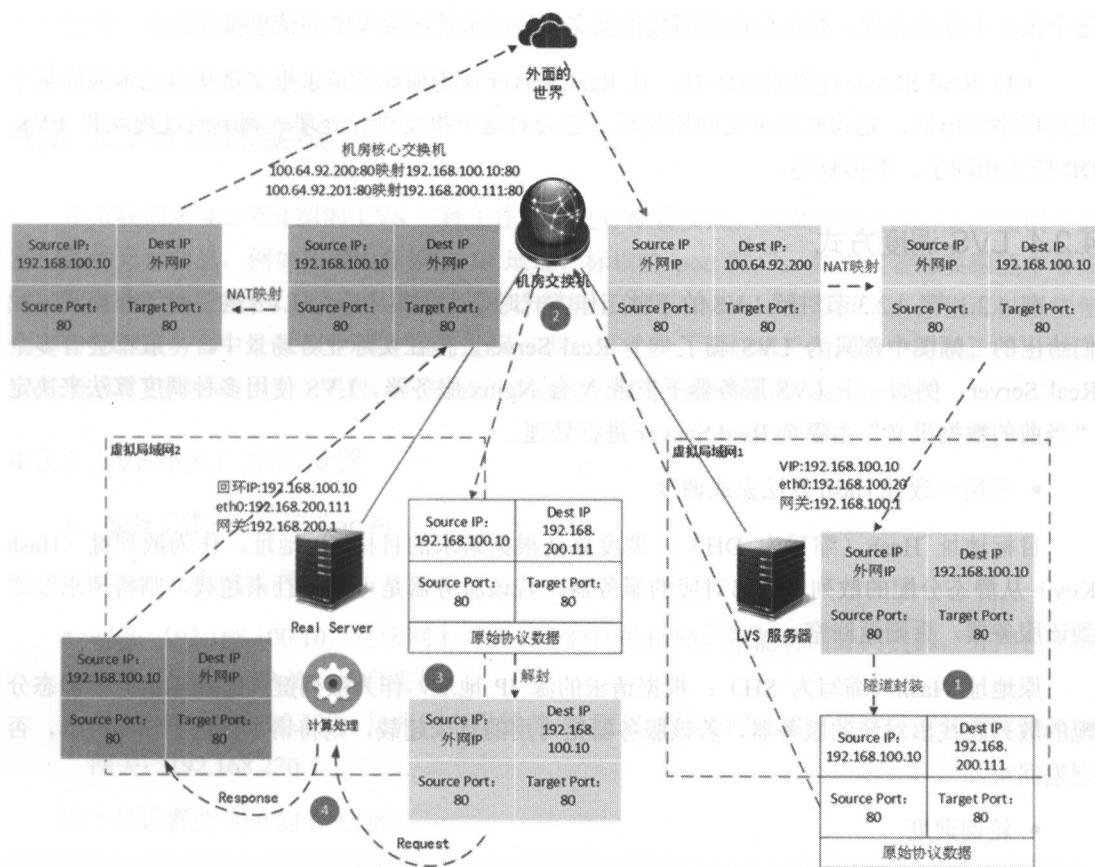


图 4-8 TUN 工作模式

图 4-8 中的连接线比较多，容易看花眼，不过只需要关注关心“有箭头的虚线”就可以了。图 4-8 中 TUN 工作模式的工作过程为：

(1) 一旦 LVS 节点发现来自目标为 192.168.100.10 VIP 的请求，就会使用 IP/IP 隧道协议对这个请求报文进行封装。而不是像 LVS-DR 模式那样重写数据报文的 Mac 信息。如果配置了多个 Real Server，那么 LVS 会使用设置的调度算法确定一个 Real Server（这里为了简单，就只画了一个 Real Server 节点）。

(2) 重新封装后的 IP/IP 隧道协议报文会重新被回发到路由器，路由器（或三层交换机）会根据设置的 LVAN 映射情况，找到目标服务器，并将这个 IP/IP 隧道报文发送过去。

(3) Real Server 收到这个 IP/IP 隧道报文后，会将这个报文进行解包。这里注意一下，一般情况下 IP/IP 隧道报文会进行分片，就如同 IP 报文分片一样，只是为了讲解方便。我们假定

这个报文不需要分片，解压后得到的数据报文就是原来发送给 VIP 的请求报文。

(4) Real Server 设置的回环 IP，让 Real Server 认为原始的请求报文是从自己本地的某个应用程序发出的，完成原始报文的校验后，它会对这个报文进行处理。剩下的过程就和 LVS-DR 模式相同了，不再复述。

#### 4.2.4 LVS 调度方式

在 4.2.1 节~4.2.3 节讲解 LVS 的三种工作方式时，为了集中介绍 LVS 的三种工作模式，我们给出的三幅图中都只为 LVS 画了一个 Real Server。但在实际业务场景中，一般都会有多个 Real Server：例如一个 LVS 服务器下拖带  $N$  台 Nginx 服务器。LVS 使用多种调度算法来决定“当前的数据报文”由哪个 Real Server 进行处理。

- 利用一致性 Hash 算法完成调度

目标地址 Hash（缩写为 DH）：调度算法根据请求的目标 IP 地址，作为散列键（Hash Key）从静态分配的散列表找出对应的服务器，若该服务器是可用的且未超载，则将请求发送到该服务器，否则返回空。

原地址 Hash（缩写为 SH）：根据请求的源 IP 地址，作为散列键（Hash Key）从静态分配的散列表找出对应的服务器，若该服务器是可用的且未超载，则将请求发送到该服务器，否则返回空。

- 轮询调度

最简轮询（缩写为 RR）：调度算法将外部请求按顺序轮流分配到集群中的真实服务器上，它均等地对待每一台服务器，而不管服务器上实际的连接数和系统负载。

最少连接轮询（缩写为 LC）：请注意“最少连接轮询”和“最少连接加权轮询”两种调度算法的区别。调度器通过“最少连接”调度算法动态地将网络请求调度到已建立的连接数最少的服务器上。注意请求肯定会被分配到这台目前连接数最少的 Real Server 上面，不会考虑几率问题。

- 加权轮询调度

性能加权轮询（缩写为 WRR）：调度算法根据真实服务器的不同处理能力来调度访问请求。这样可以保证处理能力强的服务器能处理更多的访问流量。调度器可以自动询问真实服务器的负载情况，并动态地调整其权值。

最少连接数的加权轮询（缩写为 WLC）：具有较高权值的服务器将承受较大比例的活动连接负载。调度器可以自动询问真实服务器的负载情况，并动态地调整其权值。注意，是按照

一个比例，有较高的分配几率，而不是像 LC 一样“肯定分配”。

## 4.3 LVS 设置实战

本节我们对 4.2 节介绍的 LVS 三种工作方式进行配置实践，实践中将使用 4.2 节图例中提到的各种元素信息，例如图例中标识的 IP 地址、标识的 port 端口。如果读者想要跟随本书描述的配置步骤进行操作，那么可以在自己的计算机上使用 VMware 虚拟机（当然也可以直接使用多台物理服务器），按照下面介绍的过程一步一步实践。我们将采用两台虚拟机，一台作为 LVS 节点（LVS Server），另外一台安装了 Nginx 的服务器作为 Real Server 节点。

### 4.3.1 LVS-NAT 方式设置

#### 1. 准备工作——LVS Server

LVS Server 有两张网卡。

- eth0: 192.168.100.10，这张网卡对应一个封闭的内网，不能访问外网资源，外网也不能直接通过这个 IP 访问这台主机。
- eth1: 192.168.220.100，这张网卡设置的 IP 可以访问外网，也可以被外网访问。eth1 的网关: 192.168.220.1。

以下是设置的 eth0 的 IP 信息：

```
[root@lvs1 ~]# cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE="eth0"
BOOTPROTO="static"
HWADDR="00:0C:29:3E:4A:4F"
ONBOOT="yes"
TYPE="Ethernet"
IPADDR="192.168.100.10"
NETMASK="255.255.255.0"
```

以下是设置的 eth1 的 IP 信息：

```
[root@lvs1 ~]# cat /etc/sysconfig/network-scripts/ifcfg-eth1
DEVICE="eth1"
BOOTPROTO="static"
HWADDR="00:0C:29:3E:4A:59"
ONBOOT="yes"
TYPE="Ethernet"
IPADDR="192.168.220.100"
NETMASK="255.255.255.0"
```



```
GATEWAY="192.168.220.1"
```

以上是最基本的 IP 设置过程，在这里就不进行参数的赘述了。记得设置完成后，要重启 network 服务：

```
[root@lvs1 ~]# service network restart
```

启动完成后可以使用 ping 命令检查连通性，证明到外网的网关工作是正常的。另外还可以通过 route 命令检查：

```
[root@lvs1 ~]# route
Kernel IP routing table
```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
192.168.100.0	*	255.255.255.0	U	1	0	0	eth0
192.168.220.0	*	255.255.255.0	U	1	0	0	eth1
default	192.168.220.1	0.0.0.0	UG	0	0	0	eth1

注意，route 表中有一项 eth1 的 default 的信息，指向 192.168.220.1。说明路由配置是正确的。

## 2. 准备工作——Real Server

Real Server 有一张网卡，在一个封闭的内网环境中。注意，Real Server 没有必要直接访问外网，其中可能的原因是由于安全性的考虑，不允许任何 Real Server 节点直接访问外网；也有可能是因为外网 IP 数量的限制，不可能做到每一个服务器节点都绑定一个外网 IP。

eth0:192.168.100.11，这样 LVS Server 和 Real Server 就组成了一个相对封闭的局域网络。注意按照我们介绍的 NAT 原理，Real Server 的 eth0 的默认网关要设置成 LVS Server：192.168.100.10。

在 Real Server 上运行了一个 Nginx 程序，在 80 端口上。这样方便在后续的过程中，测试 LVS-NAT 的工作是否正常。

以下是设置好的 Real Server eth0 的 IP 信息：

```
[root@vml ~]# cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE="eth0"
BOOTPROTO="static"
HWADDR="00:0C:29:45:04:32"
ONBOOT="yes"
TYPE="Ethernet"
IPADDR=192.168.100.11
NETMASK=255.255.255.0
GATEWAY="192.168.100.10"
```

一定注意：Real Server 的网关要设置到 LVS 的 IP：192.168.100.10。接着直接访问



192.168.100.11 这个 IP，若 Nginx 工作是正常的，那么读者就可以在浏览器上看到类似如图 4-9 所示的显示信息。

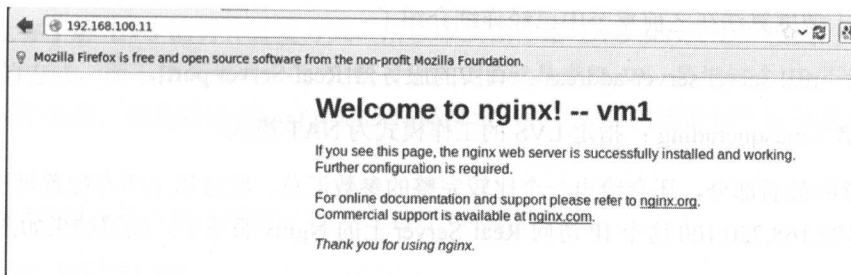


图 4-9 NAT 检查 Real Server 工作状况

完成准备工作后，我们就可以开始安装和配置 LVS-NAT 工作方式了。

### 3. 开始安装和配置 LVS-NAT 方式

ipvsadm 是一个 LVS 的管理程序。在本示例中我们对 LVS 的配置都是通过这个管理程序进行实现的（在实际工作环境下，一般也是这样做的）。首先我们要安装 ipvsadm：

```
[root@lvs1 ~]# yum -y install ipvsadm
// 这里略去了 yum 命令运行时所显示的日志
```

下面设置 LVS 机器支持 IP 转发功能（默认 IP 转发功能是关闭的，重启机器后，又会关闭）。注意：如果你使用 vim 或者 vi 命令改写文件，那么是不会成功的。因为在 Linux 系统上控制 IP 转发功能的文件存在于内存，不在硬盘上。所以只能通过 echo 这样的命令改写。

```
[root@lvs1 ~]# echo 1 >> /proc/sys/net/ipv4/ip_forward
```

设置完成后，查看是否改写成功：

```
[root@lvs1 ~]# cat /proc/sys/net/ipv4/ip_forward
1
```

如果以上 cat 命令显示的是“1”，那么表明设置成功了。最后执行如下命令，对 LVS 的工作模式进行设置：

```
[root@lvs1 ~]# ipvsadm -At 192.168.220.100:80 -s rr
[root@lvs1 ~]# ipvsadm -at 192.168.220.100:80 -r 192.168.100.11 -m
```

解释一下以上 ipvsadm 命令使用的参数。

-A 或者--add-service：在内核的虚拟服务器表中添加一条新的虚拟服务器记录。也就是增加一台新的虚拟服务器。

- t 或者 --tcp-service service-address: 说明虚拟服务器提供的是 tcp 的服务。
- s 或者 --scheduler scheduler: 使用的调度算法, 可选项包括: rr|wrr|lc|wlc|blc|blcr|dh|sh|sed|nq, 关于调度算法在之前章节中已经详细介绍了。
- r 或者 --real-server server-address: 真实的服务器[Real-Server:port]。
- m 或者 --masquerading: 指定 LVS 的工作模式为 NAT 模式。

在本章的最后部分, 还会给出一个比较完整的参数汇总。经过以上所有配置过程, 我们就可以通过 192.168.220.100 这个 IP 访问 Real Server 上的 Nginx 服务了, 配置结果如图 4-10 所示。

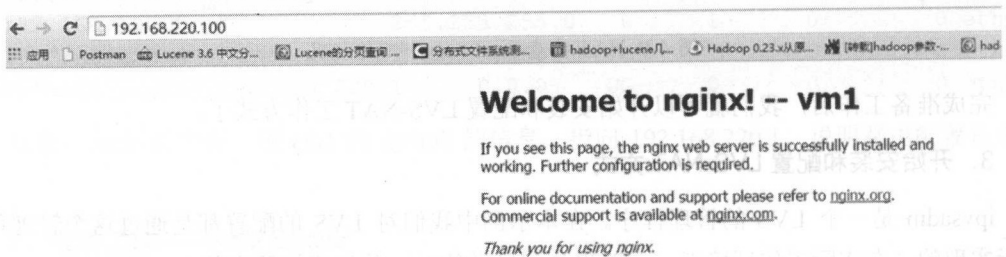


图 4-10 NAT 配置结果

#### 4. 关于 iptables 和重启服务器的说明

在配置 LVA-NAT 的过程中, 建议关闭 LVS 和 Real Server 的防火墙功能。这样可以避免不必要的错误发生, 增加一次配置成功的几率。但是正式生产环境中, LVS 的防火墙根据实际情况最好还是要打开的。

请注意, 刚才使用 ipvsadm 配置的信息, 在 LVS 服务器重启后, 就会失效, 包括 ip\_forward 的配置。所以最好制作一个脚本文件, 并加入到/etc/profile 中:

```
[root@lvs1 ~]# vim /usr/lvsshell.sh
#!/bin/bash
echo 1 > /proc/sys/net/ipv4/ip_forward
ipvsadm -C
ipvsadm -At 192.168.220.100:80 -s rr
ipvsadm -at 192.168.220.100:80 -r 192.168.100.11 -m
```

### 4.3.2 LVS-DR 模式设置

#### 1. 准备工作——LVS Server

为了让你了解 LVS 的另外设置方式，所以本次在介绍 LVS-Server DR 模式的设置过程时，使用 VIP 的方式，而不是两张网卡的方式。所谓 VIP 就是虚拟 IP，是指这个 IP 不会固定捆绑到某一个网卡设备，而是通过 `ifconfig` 命令按需绑定，并且在“适当时候”这种绑定关系会随之变化。

DIP (LVS 服务节点的内网固定 IP) : 192.168.220.137

VIP: 192.168.220.100

首先我们看看在没有设置 VIP 之前，LVS 主机上的 IP 信息：

```
[root@lvs1 ~]# ifconfig
eth1      Link encap:Ethernet  HWaddr 00:0C:29:3E:4A:59
          inet addr:192.168.220.137  Bcast:192.168.220.255  Mask:255.
255. 255.0
          inet6 addr: fe80::20c:29ff:fe3e:4a59/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2612 errors:0 dropped:0 overruns:0 frame:0
          TX packets:117 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:159165 (155.4 KiB)  TX bytes:8761 (8.5 KiB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:210 errors:0 dropped:0 overruns:0 frame:0
          TX packets:210 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:16944 (16.5 KiB)  TX bytes:16944 (16.5 KiB)
```

以上信息很清楚了，不需要再进行额外的说明。然后在以上这个节点设置一个 VIP（虚拟 IP）信息。

以上命令中的“eth1:0”表示这个 VIP 绑定的目标网卡设备，“192.168.220.100”这个 IP 就是 VIP 的值，广播地址为“192.168.220.100”，子网掩码为“255.255.255.255”，“up”关键字表示立即启动这个 VIP。接着我们还要通过 `route` 命令，在路由表上添加对这个 VIP 的路由信息。

设置完成后，我们可以在一个外网节点上，通过 `ping` 命令，检查这个 VIP 的连通性：

```
# 这是网络中存在的另一个 DOS 操作系统，它可以看成一个外网的节点
```

```
[root@lvs1 ~]# ifconfig eth1:0 192.168.220.100 broadcast 192.168.220.100 netmask 255.255.255.255 up
[root@lvs1 ~]# route add -host 192.168.220.100 dev eth1:0
C:\Users\yinwenjie> ping 192.168.220.100
```

正在 Ping 192.168.220.100 具有 32 字节的数据:

来自 192.168.220.100 的回复: 字节=32 时间<1ms TTL=64

来自 192.168.220.100 的回复: 字节=32 时间<1ms TTL=64

从以上 Ping 命令的执行情况看, 这个 DOS 操作系统基于 LVS 主机上放置的 VIP 连接成功。这样我们就完成了 LVS 主机设置 LVS-DR 工作模式的准备工作。注意:

- 读者设置过程中, 防火墙最好关闭, 但在正式生产环境中, LVS-Server 的防火墙则最好打开。
- VIP 信息在 LVS 主机重启后会消失。所以读者最好将设置 VIP 的命令做成一个脚本。

## 2. 准备工作——Real Server

RIP: 192.168.220.132

真实服务器的准备工作, 需要保证真实服务器能够访问外网网关, 并且保证由 LVS 改写的报文能够被 Real Server 处理 (请参见本章介绍 LVS 原理的内容部分: 4.1 和 4.2 节), 这就需要做一个回环 IP。首先我们查看一下, 还没有开始设置之前的 IP 信息:

```
[root@localhost ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:FC:91:FC
          inet  addr:192.168.220.132    Bcast:192.168.220.255    Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe80:91fc/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2384 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1564 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1551652 (1.4 MiB)  TX bytes:144642 (141.2 KiB)

lo        Link encap:Local Loopback
          inet  addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:44 errors:0 dropped:0 overruns:0 frame:0
          TX packets:44 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3361 (3.2 KiB)  TX bytes:3361 (3.2 KiB)
```

这台 Real Server 上有一个 Nginx 以保证我们观测 LVS-DR 方式的运行情况。正常情况下在外网使用 192.168.220.132 这个 IP，能够访问到 Nginx 的页面（因为这个 Real Server 节点能工作在 LVS-DR 模式下的一个前提条件，就是它可以直接被外网访问），如图 4-11 所示。

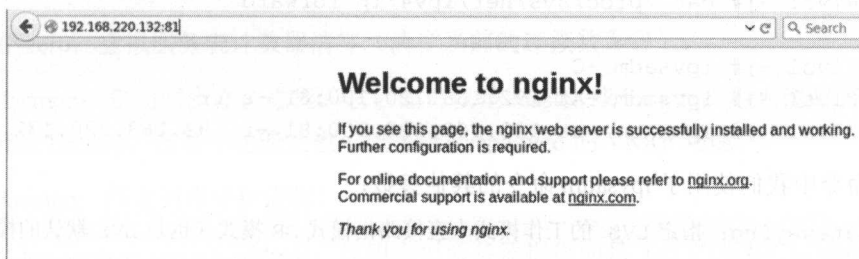


图 4-11 DR 检查 Real Server 工作状态

接下来，开始设置 Real Server 上的回环 IP，首先关闭这台机器进行 ARP 查询的功能，否则，Real Server 会在路由器或者交换机上去查询 192.168.220.100 这个 IP 对应的 Mac 地址（注意，以下的信息在重启后都会被还原）：

```
echo "1" >/proc/sys/net/ipv4/conf/lo/arp_ignore
echo "2" >/proc/sys/net/ipv4/conf/lo/arp_announce
echo "1" >/proc/sys/net/ipv4/conf/all/arp_ignore
echo "2" >/proc/sys/net/ipv4/conf/all/arp_announce
```

接着可以设置回环 IP 了：

```
[root@vm3 ~]# ifconfig lo:0 192.168.220.100 broadcast 192.168.220.100
netmask 255.255.255.255 up
[root@vm3 ~]# route add -host 192.168.220.100 dev lo:0
```

以上命令的主要意义在之前介绍过了，这里同样不再赘述。最后检查新的路由信息：

```
[root@vm3 ~]# route
Kernel IP routing table
```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
192.168.220.100	*	255.255.255.255	UH	0	0	0	lo
192.168.220.0	*	255.255.255.0	U	1	0	0	eth0
default	.168.220.1	0.0.0.0	UG	0	0	0	eth0

以上设置完成后，需要通过 ping 命令检查一下网关是否可用（最好用 ping 命令检查一下外网的某个地址的连通性，例如 163.com）。在 LVS-DR 模式下，Real Server 是直接向请求方返回结果，所以一定要保证网关可用。

### 3. 开始配置 LVS-DR 模式

在验证了 LVS Server 和 Real Server 都准备好以后，就可以进行 LVS-DR 模式的设置了。



LVS 管理软件 `ipvsadm` 的安装就不再复述了，在之前讨论 LVS-NAT 模式的设置方法时已经介绍过了。

```
[root@lvs1 ~]# echo 1 > /proc/sys/net/ipv4/ip_forward
[root@lvs1 ~]# cat /proc/sys/net/ipv4/ip_forward
1
[root@lvs1 ~]# ipvsadm -C
[root@lvs1 ~]# ipvsadm -At 192.168.220.100:81 -s rr
[root@lvs1 ~]# ipvsadm -at 192.168.220.100:81 -r 192.168.220.132 -g
```

以上命令中我们使用了 `ipvsadm` 命令的其他参数：

`-g -gatewaying`：指定 LVS 的工作模式为直接路由模式 DR 模式（也是 LVS 默认的模式）

最后就可以在外网，通过 192.168.220.100 这个 IP 访问 192.168.220.132 这台 Real Server 上的 Nginx 服务器了，如图 4-12 所示。



图 4-12 DR 工作效果

### 4.3.3 ipvsadm 参数汇总

`-A` 或者 `-add-service`：在内核的虚拟服务器表中添加一条新的虚拟服务器记录。也就是增加一台新的虚拟服务器。

`-E` 或者 `-edit-service`：编辑内核虚拟服务器表中的一条虚拟服务器记录。

`-D` 或者 `-delete-service`：删除内核虚拟服务器表中的一条虚拟服务器记录。

`-C` 或者 `-clear`：清除内核虚拟服务器表中的所有记录。

`-R` 或者 `-restore`：恢复虚拟服务器规则。

`-S` 或者 `-save`：保存虚拟服务器规则，输出为 `-R` 选项可读的格式。

`-a` 或者 `-add-server`：在内核虚拟服务器表的一条记录里添加一条新的真实服务器记录。也就是在一个虚拟服务器中增加一台新的真实服务器。



- e 或者 -edit-server: 编辑一条虚拟服务器记录中的某条真实服务器记录。
- d 或者 -delete-server: 删除一条虚拟服务器记录中的某条真实服务器记录。
- L 或者 -list: 显示内核虚拟服务器表。
- Z 或者 -zero: 虚拟服务表计数器清零（清空当前的连接数量等）。
- start-daemon: 启动同步守护进程。它后面可以是 master 或 backup, 用来说明 LVS Router 是 master 或是 backup。在这个功能上也可以采用 keepalived 的 VRRP 功能。
- stop-daemon: 停止同步守护进程。
- t 或者 -tcp-service service-address: 说明虚拟服务器提供的是 tcp 的服务[vip:port] or [real-server-ip:port]。
- u 或者 -udp-service service-address: 说明虚拟服务器提供的是 udp 的服务[vip:port] or [real-server-ip:port]。
- f 或者 -fwmark-service fwmark: 说明是经过 iptables 标记过的服务类型。。
- s 或者 -scheduler scheduler: 使用的调度算法, 选项 rr|wrr|lc|wlc|lb|lcr|dh|sh|sed|nq, 默认的调度算法是 wlc。
- p 或者 -persistent [timeout] : 持久稳固的服务。这个选项的意思是来自同一个客户的多次请求, 将被同一台真实的服务器处理。timeout 的默认值为 300 秒。
- r 或者 -real-server server-address -: 真实的服务器[Real-Server:port]。
- g 或者 -gatewaying -: 指定 LVS 的工作模式为直接路由 (LVS-DR) 模式 (也是 LVS 默认的模式)。
- i 或者 -ipip -: 指定 LVS 的工作模式为隧道模式。
- m 或者 -masquerading -: 指定 LVS 的工作模式为 NAT 模式。
- w 或者 -weight weight -: 真实服务器的权值。
- mcast-interface -: 指定组播的同步接口。
- connection -: 显示 LVS 目前的连接, 如: ipvsadm -L -c。
- timeout -: 显示 tcp tcpfin udp 的 timeout 值, 如: ipvsadm -L -timeout。
- daemon -: 显示同步守护进程状态。

- stats -: 显示统计信息。
- rate -: 显示速率信息。
- sort -: 对虚拟服务器和真实服务器排序输出。
- numeric -n -: 输出 IP 地址和端口的数字形式。

## 第 5 章

# 其他负载层技术

本章将概述一些本书在负载均衡部分还没有讲到的其他负载层技术。这些技术目前都是成熟的，且已经在国内外有大量成功案例，也有很多供应商直接提供技术服务，完全不需要像 LVS、Nginx、Keepalived 等组件那样需要自行配置和参数调试（当然也有很多云服务商直接在 PaaS 层向客户提供 LVS、Nginx、Keepalived 那样的技术）。

### 5.1 DNS 和智能 DNS

DNS（Domain Name System，域名解析服务）是将浏览器上输入的域名解析成具体 IP 地址的关键技术。那么读者可能就会问了，如果这个技术的主要工作是将域名解析成对应的服务器 IP 地址，那它和负载均衡有什么关系呢？

是的，单从对 DNS 技术的文字描述很难看出来它和负载均衡之间的关系。但如果我们将 DNS 中域名和 IP 地址的对应关系从一个域名对应一个服务器 IP 扩展到一个域名对应多个服务器 IP 呢（图 5-1）？

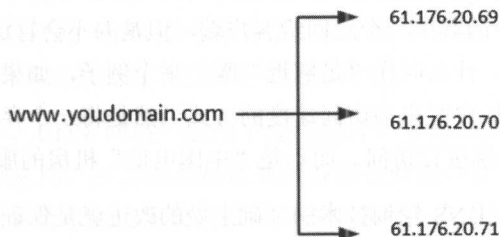


图 5-1 DNS 轮询

当用户 A 询问域名对应的 IP 时, DNS 服务向用户返回的 IP 是 61.176.20.69; 当用户 B 询问域名对应的 IP 时, DNS 服务向用户返回的 IP 是 61.176.20.70; 当用户 C 询问域名对应的 IP 时, DNS 服务向用户返回的 IP 则是 61.176.20.71。

这样在 DNS 服务内就形成了一种负载均衡形式, 称为 DNS 轮询。目前绝大多数 DNS 服务商提供的服务都支持 DNS 轮询, 且可靠稳定。但是在实际工作中, 很少有互联网服务单纯利用 DNS 轮询技术构建负载层技术, 这是因为 DNS 轮询技术也存在一些不足:

- 单纯使用 DNS 轮询容易导致各服务节点压力不平衡。

之所以产生这个缺陷是因为 DNS 轮询所分配的 IP 会缓存在客户端上, 并且 DNS 轮询并不能检测 IP 对应的各服务器当前的处理压力。所以一段时间后, DNS 并不能知晓所有客户端的在线状态, 也无法知晓各 IP 对应的服务节点的服务压力。这时就可能出现连接到 X 服务节点上的客户端有 500 个, 而连接到 Y 服务节点上的客户端只有 100 个 (因为一部分客户端已经下线了); 也可能出现尽管某个服务节点的处理压力已经很大了, 但是 DNS 轮询后依然将新的客户端分配给这台服务器处理的情况。

- 当某个 IP 由于各种原因不能提供服务时, DNS 上的同步时间会非常慢。

DNS 服务按照从上自下的树形结构进行设计, 也就是说上层 DNS 服务中域名和 IP 的对应关系发生变化后, 它需要一定的时间将变化同步到下层的 DNS 服务中。这个同步时间一般在 0.5 小时到 6 小时之间 (甚至会需要更长时间, 这要看服务商所提供的 DNS 服务在互联网中的位置等因素), 而这段时间内会有两种错误情况发生: 当本地缓存了故障 IP 节点信息的客户端访问域名时, 会直接访问到这个不能再提供服务的 IP 地址, 导致错误; 当本地没有缓存 IP 节点的新客户端请求域名解析时, 后者可能会将这个故障的 IP 节点返回给客户端进行访问。

基于以上这些原因的考虑, DNS 轮询技术一般和 LVS 技术联合使用, 当然也有 DNS 轮询和 Nginx 联合使用的情况。总之, 单独使用 DNS 轮询技术构建系统的负载层的方案, 在实际应用中较少见。

DNS 轮询技术还有一种改进功能就是 DNS 智能解析, 上面介绍的 DNS 轮询技术可以将若干个预先设置的 IP 地址中的其中一个返回给客户端, 但是并不会管这个 IP 地址是否正常或者是否离客户端“足够近”。什么叫作“足够近”呢? 举个例子, 如果客户端是通过“中国移动”接入互联网的, 那么直接加快客户端访问速度的方式当然是将一个存在于“中国移动”机房的服务器 IP 地址提供给客户端进行访问, 而不是“中国电信”机房的服务器 IP。

DNS 智能解析技术在 DNS 轮询技术的基础上做的改进就是保证返回给客户端的域名对应的服务器 IP, 都是尽可能离用户“足够近”的。它的原理也很简单, 即是在 DNS 服务器上有一个预先设置的 IP 划区表 (这个 IP 划区表/IP 地址库可以在“淘宝网”上购买到, 只不过准确

度可能差些)，当客户端发送 DNS 解析请求时，DNS 服务器会识别出当前客户端 IP 的行政区域和互联网接入商的信息，然后就可以从预先设置的若干个服务器 IP 中找到离它最近的一个返回给客户端。

## 5.2 CDN 网络

CDN (Content Delivery Network, 内容分发网络)，按照目前的技术限制条件，无论读者是通过新兴的虚拟技术搭建业务系统，还是使用传统的物理主机托管的方式搭建业务系统。出口带宽都是要用钱买的，很显然总带宽只有 20Mbps 是非常节约资金的，但是基本上无法满足大量用户同时访问的要求；如果购买总带宽为 20Gpbs，倒是可以满足多数情况下大量用户同时访问所需的带宽要求，但公司/技术团队将为这 20Gpbs 的出口带宽每月支付高昂的费用，并且访问效果还不一定好。

为什么访问效果不一定好呢？这是因为国内有三家主要的互联网接入服务商：联通、电信和移动。而这三家互联网接入商的互通一般都需要经过多层路由，读者可以想象一下通过联通接入互联网的一个客户端要访问电信机房内某个服务器上的一张图片时，需要经过多少层路由才能实现访问。

为了解决这个严重的问题，聪明的 IT 工程师提出了“资源访问的最后一公里”概念。也就是说虽然真实的服务节点放在电信机房，但是通过联通网络接入的客户端在访问时并不会到很远的电信机房访问这个资源，而到一个已经预先把将要访问的资源进行了缓存且离访问者最近的某个联通机房的服务节点上访问资源。

这种方式既缓解了主服务的机房过大的带宽需求，又优化了各个接入商对同一资源的访问效果。目前国内外有大量的供应商提供稳定的 CDN 服务，这些服务的可用性基本可以保证 99.999% (5 个 9 的可用性)，并且收费合理。所以如果读者所在公司/团队没有特别的需求，笔者并不建议由读者所在公司/团队自行搭建 CDN 服务。这里提到的“不建议”的原因还有一个，就是 CDN 服务是否快速、稳定、高效的重要评判指标是边界节点的数量和分布情况，好的 CDN 服务商往往都为自己的 CDN 服务准备了上百个边界节点，如果读者所在公司/团队准备自建 CDN 服务，那么部署这些边界节点将花费更多资金。

在互联网技术实践中，CDN 网络一般用于缓存图片、多媒体文件和各种静态文件资源。这些资源的请求是可以预见的，并且也是最耗费服务器外网出口资源的。注意，CDN 网络并不负责处理业务请求，业务请求还是需要直接在业务服务器上完成的。简单的 CDN 网络也很好搭建，使用智能 DNS + 缓存组件 (例如 Squid) 就可以完成。智能 DNS 主要用于识别客户

端 IP，并且找到离客户端最近的服务节点；缓存组件作为边界节点，它负责从真正的服务节点获取资源，并缓存在本地（图 5-2）。

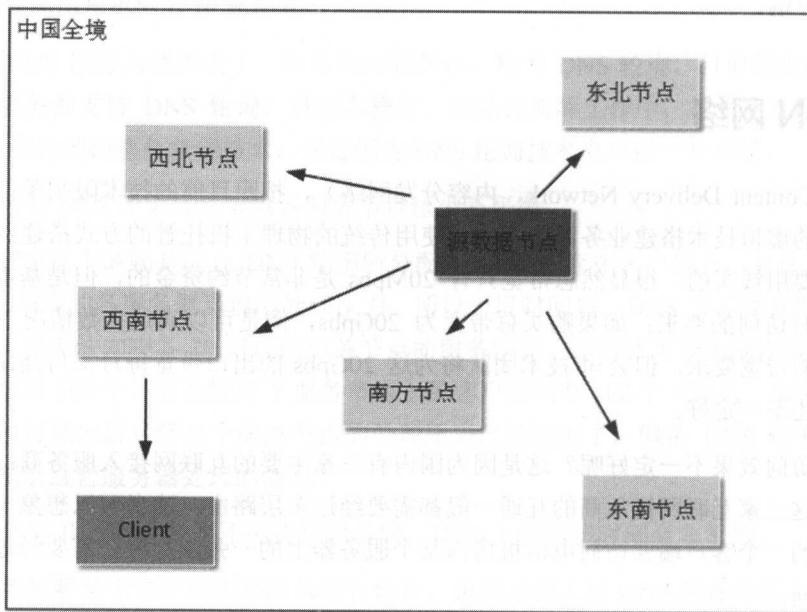


图 5-2 星型 CDN 结构

当客户端请求一张图片资源时，CDN 服务会首先解析域名信息（可能是顶级域名也可能是二级域名），CDN 会将离客户端“最近”的服务节点 IP 返回给客户端，这个服务 IP 对应了一个边界节点。接下来客户端就会向这个边界节点正式请求图片资源。边界节点对请求的处理又分为两种情况：如果边界节点存在这个图片资源，则直接返回给客户端；如果边界节点没有这个图片资源，则会向真实的服务节点请求这个资源，缓存到本地后再返回给客户端。CDN 边界节点对资源的缓存可以使用 LRU 算法，也可以固定一个过期时间，这完全取决于业务形态和资源性质。

如果读者所在的公司/团队需要自建 CDN 服务，那么以上介绍的这种 CDN 网络结构就是可以完成最快速搭建的第一选择，实际上这种结构也是目前大多数 CDN 服务商提供的产品架构方式。但是这种 CDN 网络存在一个很明显的问题，就是真实服务节点的压力还是比较大。想象一下如果客户端请求的是一个 720P 的电影视频，怎么办？难道所有 100 个边界节点都从真实服务节点上读取数据吗？那样的话真实服务节点的带宽资源再大、CPU 性能再好、I/O 读写能力再强也是不够用的。这时我们就需要另一种，可以将所有边界节点的网络能力、计算能力都调用起来的改进方案了（图 5-3）。



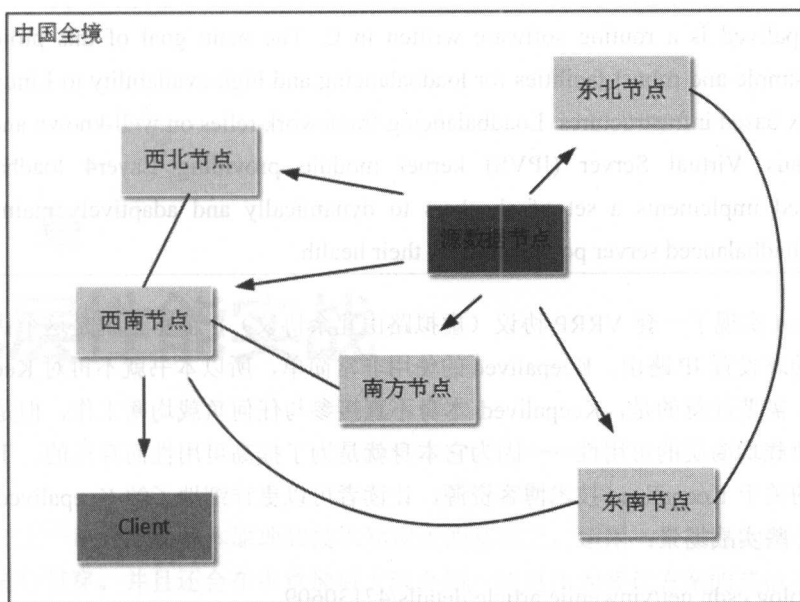


图 5-3 网状 CDN 结构

在这种结构下，某个边界节点要获取新的文件资源时，并不一定会到主服务节点上进行请求。它首先会到索引服务器上寻找这个文件资源所在的另一个（或多个）边界节点，并从这个（或多个）边界节点上获取文件资源。如果当前资源并不存在于任何其他的边界节点上，那么当前的节点才会到主服务节点上获取资源。

为什么某个节点会同时从多个边界点上获取同一个文件呢？这是因为网状 CDN 结构中，某一个大文件被拆分为多个片段，并由索引服务负责记录它们的位置。为了加快文件的获取速度，这个获取资源的节点会同时从不同的节点上分别取得文件的不同片段，最后在本地形成一个完整的文件。

这种网状 CDN 结构非常适合对大文件进行交换，例如视频文件。实际上优酷、土豆、腾讯、迅雷等视频站点就利用了这样的 CDN 结构进行大文件传输，并且再配合客户端与客户端之间的 P2P 技术，基本上就可以让大多数视频播放满足网络传输速度的要求了，同时这也使服务节点的压力，特别是真实存储大文件的服务节点的压力降到最低。

## 5.3 Keepalived

以下文字来源于 Keepalived 官方网站：

Keepalived is a routing software written in C. The main goal of this project is to provide simple and robust facilities for loadbalancing and high-availability to Linux system and Linux based infrastructures. Loadbalancing framework relies on well-known and widely used Linux Virtual Server (IPVS) kernel module providing Layer4 loadbalancing. Keepalived implements a set of checkers to dynamically and adaptively maintain and manage loadbalanced server pool according their health.

Keepalived 实现了一套 VRRP 协议（虚拟路由冗余协议），简单来说是这个协议允许物理服务器能够动态设置 IP 路由。Keepalived 的使用非常简单，所以本书就不再对 Keepalived 进行单独介绍了。需要注意的是，Keepalived 本身不直接参与任何负载均衡工作，但是可以显著增加软件系统负载均衡层的可用性——因为它本身就是为了提高可用性而存在的。下面提供了一些笔者原创的关于 Keepalived 技术博客资源，让读者可以更详细地了解 Keepalived 技术，这些资源中还有一些实战场景：

- <http://blog.csdn.net/yinwenjie/article/details/47130609>
- <http://blog.csdn.net/yinwenjie/article/details/47211551>

## 5.4 不得不提的 Tengine

Tengine 是由“淘宝网”发起并主导的 Web 服务器项目。它在 Nginx 的基础上，针对高访问量网站的需求，添加了很多高级功能和特性。Tengine 的性能和稳定性已经在大型的网站如“淘宝网”、“天猫”等得到了很好的检验。它的最终目标是打造一个高效、稳定、安全、易用的 Web 平台（<http://tengine.taobao.org/>）。

笔者建议读者根据业务的实际情况，在生产环境中适时引入 Tengine。但在笔者对本书进行整理时，Tengine 的 2.X 版本还不稳定，所以建议使用 1.5.2 的稳定版本。请记住 Tengine 就是经过升级改造后的 Nginx。

# 第 6 章

## 负载层性能实战

这一章在上一章介绍的基本原理和技术知识点的基础上，运用一个实际的例子。将上一章讲到的内容进行贯穿。并且还会在本章最后大致介绍一些可作为替代方案的其他手段。

### 6.1 负载层技术实战场景

我们知道负载均衡层的作用是“将来源于外部的处理压力通过某种规律/手段分摊到内部各个处理节点上”，而且不同的业务场景需要的负载均衡方式也是不一样的，架构师还需要综合考虑架构方案的成本、可扩展性、运维难易度等方面的问题。下面先介绍几种典型的业务场景，读者在阅读相应的解决方案之前也可以先想象一下，你会怎么架设这些场景的负载均衡层。

#### 6.1.1 负载场景一

如图 6-1 所示，这是一个国家级物流园区的货运订单和物流管理系统。在物流园区内的货运代理商、合作司机（货运车辆）、园区管理员和客服人员都要使用这套系统。每日 RUV 在 1 万人次，日 PV 在 10 万左右。甲方总经理使用这套系统的缘由，是想“试一下移动互联网对物流行业是否能起到提高效率的作用”。可以看出整个系统基本上没有访问压力，甲方对你设计的系统只有一个要求：能够保证系统稳定运行，并且兼顾后续功能和性能扩展能力。

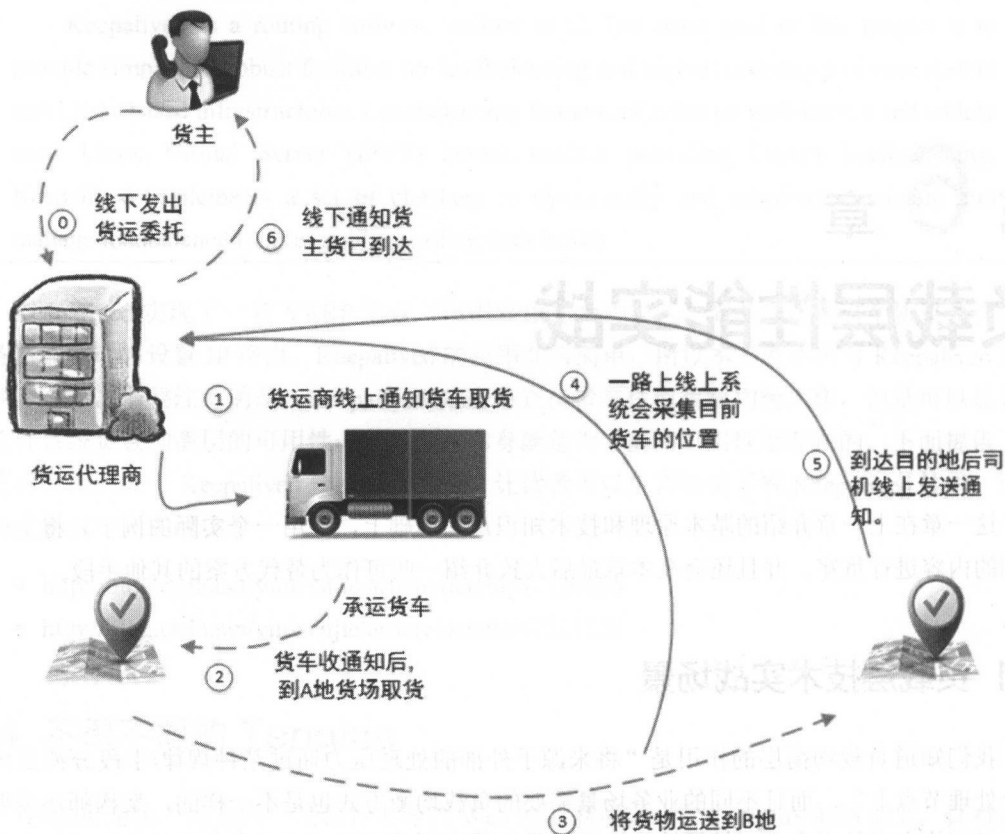


图 6-1 负载场景一

### 6.1.2 负载场景二

效果不错！在第一版系统架设后的 6 个月，货场丢货的情况大大减少，并且由于开发了货车在途情况的监控功能，货物的按时到达率也显著提升。货车司机也反映由于整个货场货车信息都是共享的，货车的待货时间明显缩短。在这期间物流园中越来越多的货运代理商、货车司机都开始使用这套系统了，整个系统的访问量成线性增长。

物流园区的总经理对整个系统的作用感到满意, 决定扩大系统的使用范围, 并增加新的功能。经过讨论甲方最终决定把整套系统开放给货主: 货主可以在系统上查看货运代理商的线路报价、线上通知代理商上门取货、监控目前自己货品的运输状态、了解第三方签收情况(图 6-2)。初步估计系统的日 RUV 将达到 10 万, 日 PV 将突破 50 万。

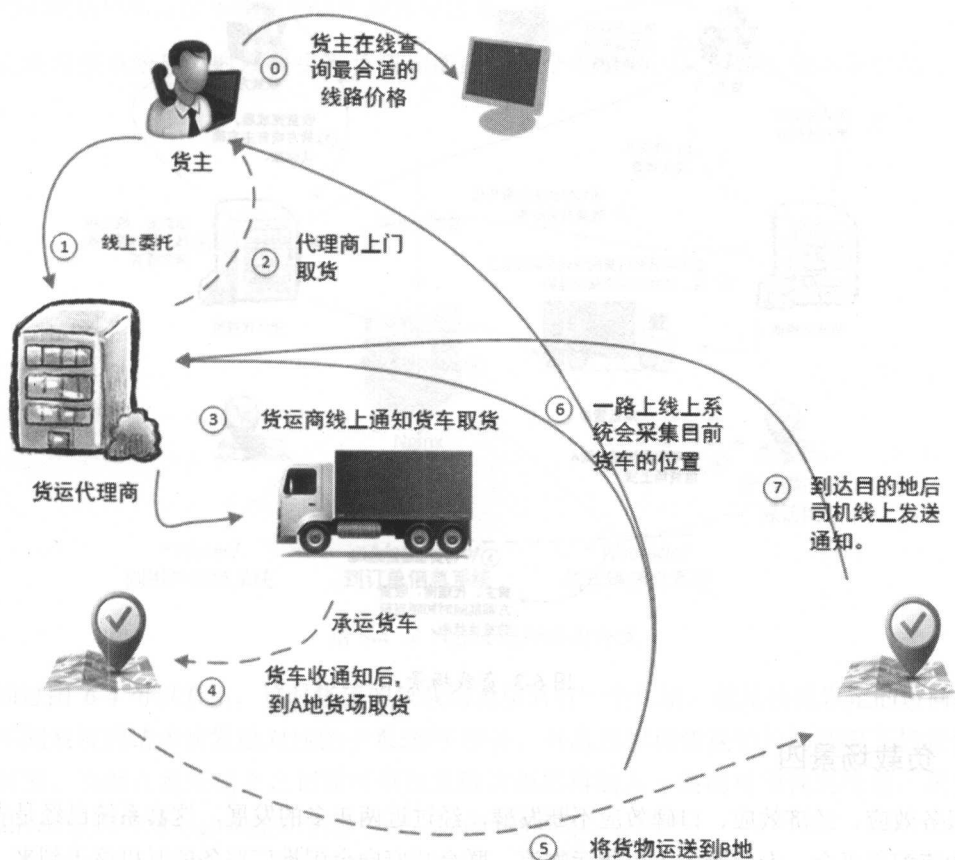


图 6-2 负载场景二

### 6.1.3 负载场景三

一年后，赞不绝口的大宗货品运输服务质量终于传到了政府领导的耳朵里。省里分管运输的领导亲自领队到物流园区参观考察，最终决定采用省政府牵头各地方政府参与的方式，将这套管理办法在整个省级范围进行推广使用。全省 10 家大型物流园和 50 家二级物流园中的上万户货运代理商、散落省内的零散代理商、10 万个人/企业货主、40 万优质车源共同接入系统。

新的功能上，增加了费用结算和运费保障功能，从货主预付款开始到第三方确认收货的整个环节都进行费用管理。为了保证线上收货环节的顺利，新版本中还增加了代理商之间的合作收货功能（图 6-3）。新系统的日 RUV 将超过 50 万，日 PV 预计将突破 250 万。

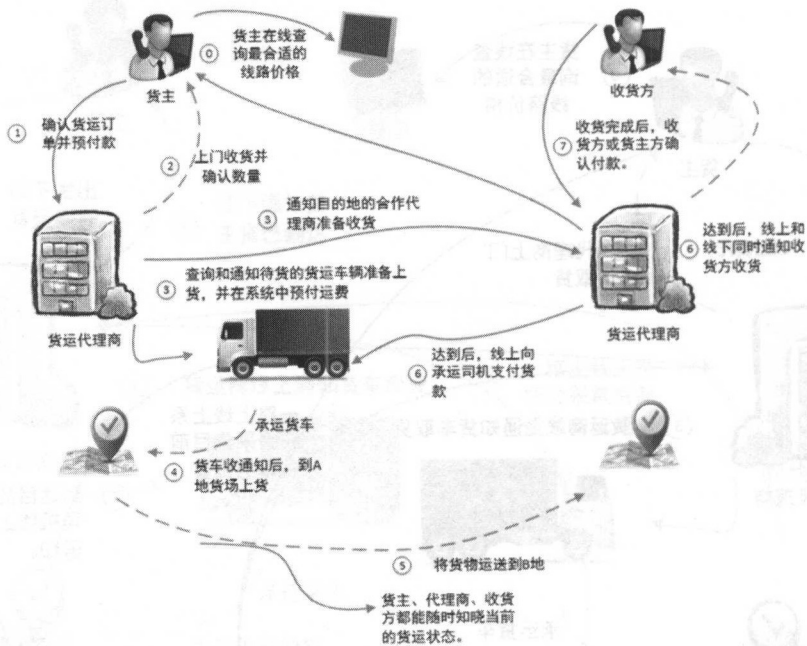


图 6-3 负载场景三

### 6.1.4 负载场景四

服务效应、经济效应、口碑效应不断发酵，经过近两年多的发展，这套系统已经是省内知名的物流配送平台，专门服务大宗货运物流。联合政府向全国推广服务的时机终于到来。预计全国 1000 多个物流园区、50 万左右物流代理商、500 万货运车辆、数不清的个人和企业货主都将使用该系统。预估的 RUV 和 PV 是多少呢？无法预估，如果按照 32 省来进行一个简单的乘法，就可以得到一个大概的值（ $50\text{万} \times 32 = 1500\text{万}$ 以上； $500\text{万} \times 32 = 1.5\text{亿}$ 以上，已经超过了 JD.com 的平峰流量），但是各省的物流业规模是不一样的，从业者数量也不一样，所以这样的预估并不科学。在这样的系统规模下我们在系统架构层面开始更多地考虑系统峰值冗余。

## 6.2 方案一：使用 Nginx 初步解决性能瓶颈问题

很显然，在第一个业务场景下，系统并没有太大的压力，它就是一套非常简单的业务系统。这个系统的日访问量也完全没有达到“有访问压力”这样的程度。但是客户有一个要求值得我们关注：要保证系统后续功能和性能扩展空间。基于这样的需求考虑，架构师在系统建立之初就需要有一个很好的业务拆分规划。例如首先会把用户信息权限子系统和订单系统进行拆分，



独立的车辆信息和定位系统可能也需要拆分出来。

这既是在系统建立时就要引入负载均衡层的一个重要原因，也是负载均衡层的重要作用之一。

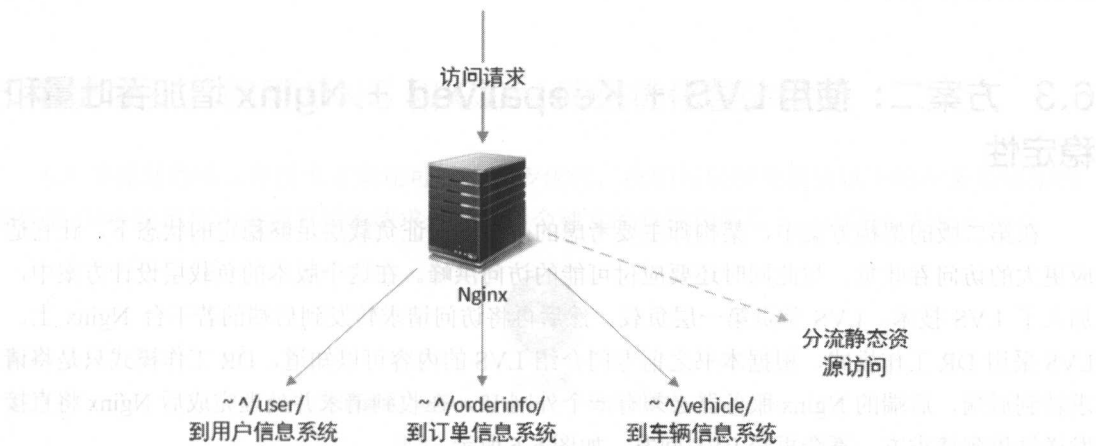


图 6-4 单 Nginx 进行路由分流

通过图 6-4 可以看出，这时搭建的负载均衡层只有一个作用，就是按照设定的访问规则将访问不同系统的请求转发给对应的子系统/子服务，并且在出现错误访问的情况下转发到错误提示页面。当然在系统建立之初就可以在负载均衡层再加入一些高可用性的考虑，例如引入 Keepalived 技术针对 Nginx 建立主备节点，如图 6-5 所示。

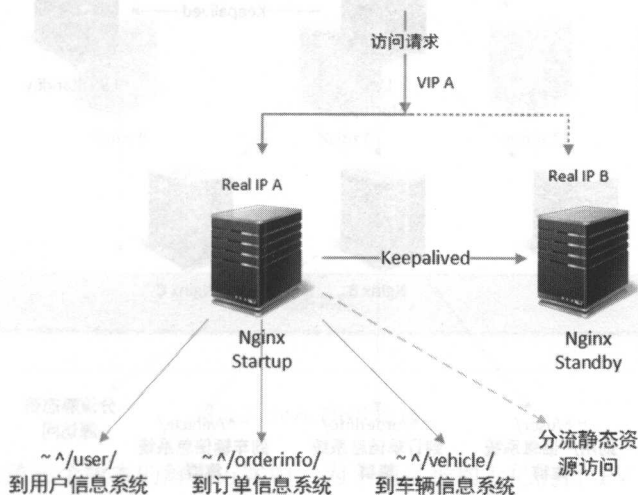


图 6-5 Nginx + Keepalived 主备方式

当然，以上两张示意图所反映出来的方案都只能算作一个方案，因为后者实际上是在前者的基础上引入了系统高可用性的设计，而负载层的吞吐总量并没有太大变化，而后者相对于前者的优点是，可以保证一个 Nginx 节点在崩溃后，另一个节点能够自动接替其工作，为工程师解决问题赢得时间。

## 6.3 方案二：使用 LVS + Keepalived + Nginx 增加吞吐量和稳定性

在第二版的架构方案中，架构师主要考虑的是：在保证负载层足够稳定的状态下，让它适应更大的访问吞吐量，与此同时还要应付可能的访问洪峰。在这个版本的负载层设计方案中，加入了 LVS 技术。LVS 负责第一层负载，然后再将访问请求转发到后端的若干台 Nginx 上。LVS 采用 DR 工作模式，根据本书之前专门介绍 LVS 的内容可以知道，DR 工作模式只是将请求转到后端，后端的 Nginx 服务器必须有一个外网 IP，在收到请求并处理完成后 Nginx 将直接发送结果到请求方，不会再经 LVS 回发，如图 6-6 所示。

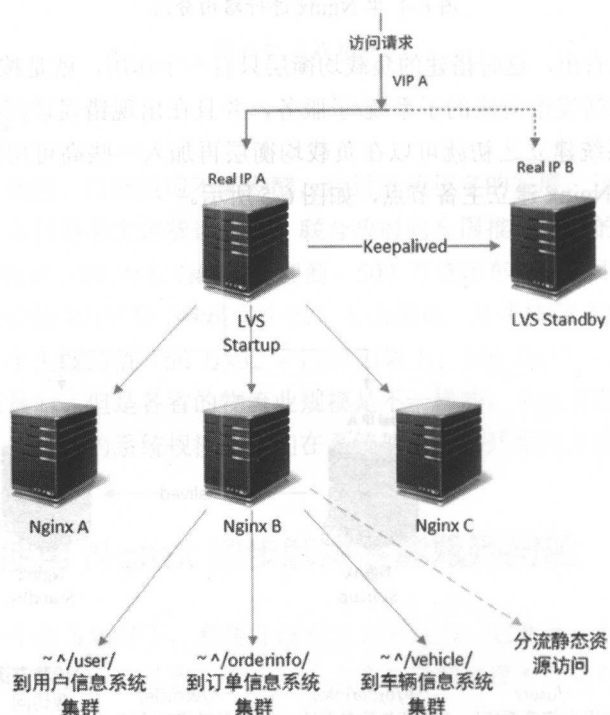


图 6-6 加入 LVS 层

Keepalived 组件在这个版本的方案中还有使用，只不过它不再保证 Nginx 节点的主备状态，而是保证两个 LVS 节点的主备状态。这是因为首先 Nginx 不再是单个节点进行负载处理，而是一个 LVS 下层包含了多台 Nginx 节点。其次 LVS 对于后端的 Nginx 节点可以进行基于端口的健康检查。

## 6.4 方案三：使用 DNS 和 CDN 网络优化整体性能

6.3 节提到的第二种技术方案还可以进一步优化，我们可以视负载层以下的业务系统架构来使用 DNS 轮询技术将客户端的请求分发到两个独立的负载均衡层上，如图 6-7 所示。

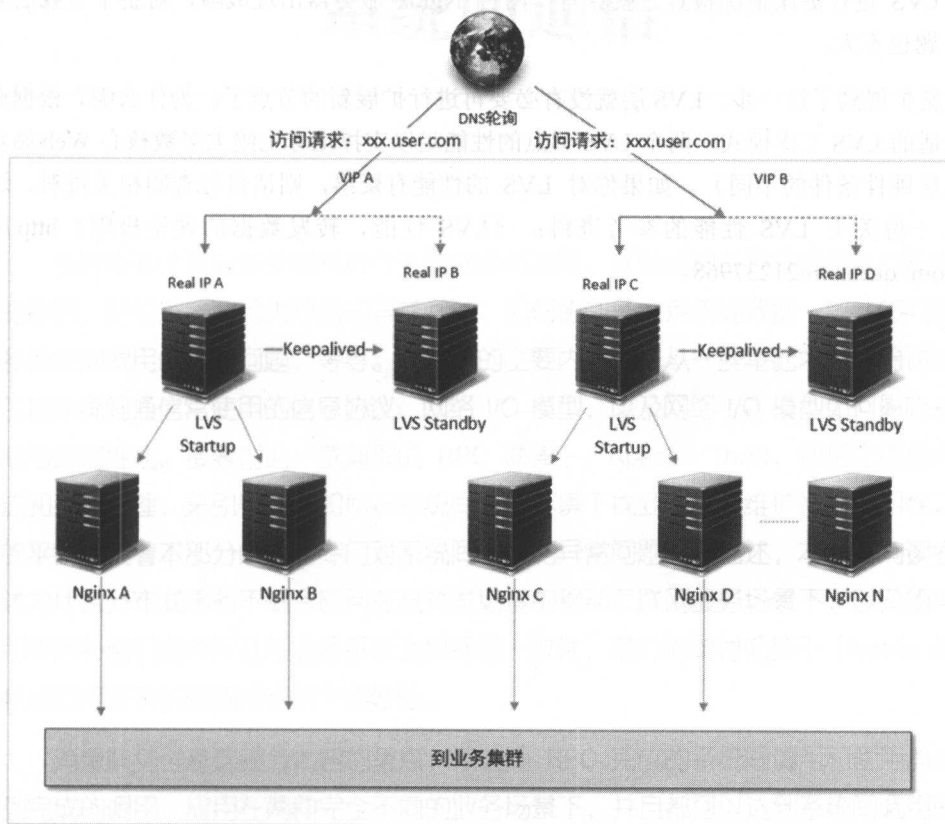


图 6-7 加入 DNS 轮询

如图 6-7 所示的第三个版本的解决方案中，为了满足平均上亿的日 PV 访问，在对业务进行行外网暴露的基础上，在互联网的最前端做了一个 DNS 轮询/DNS 智能解析。将业务系统的访

问压力首先分摊到两个对称 LVS 组上，再由每个组向下继续分拆访问压力。

- 首先我们不再像方案一、方案二中提到的那样，使用目录名分割业务系统。而是直接将业务系统的访问使用不同的二级域名进行拆分。这样的变化有利于每个业务系统都拥有自己独立的负载均衡层。
- 请注意图 6-7 技术方案中的细节：这个负载均衡层是专门为“用户信息子系统”提供负载均衡支撑的，而可能还存在的“订单子系统”、“车辆信息子系统”都会有它们独立的负载均衡层，这也是“按照业务系统拆分二级域名”的体现。
- 在 LVS 下方的 Nginx 服务，理论上可以实现无限制的扩展。同样，类似于方案二中所给出的解决方案那样，Nginx 本身不再需要 Keepalived 保持热备，而是全部交由上层的 LVS 进行健康情况检查。就算有一两台 Nginx 服务器出现故障，对整个负载层来说问题也不大。

方案扩展到了这一步，LVS 层就没有必要再进行扩展新的节点了。为什么呢？根据业务选择了合适的 LVS 工作模式，两个 LVS 节点的性能足以支撑地球上绝大多数核心 Web 站点（不考虑底层硬件条件的不同）。如果你对 LVS 的性能有疑惑，则请自行查阅相关资料。这里笔者提供一份关于 LVS 性能的参考资料：《LVS 性能，转发数据的理论极限》<http://www.zhihu.com/question/21237968>。

## 第三部分

# 系统间通信

---

系统间通信是业务系统间进行关联的必然活动，这些活动也是多数系统问题出现的原因，例如系统调用失败后的异常问题、如何协调多个系统间数据一致性的问题、多系统间调用的性能问题，等等。本部分的主要内容将先从一些基础内容开始讨论，包括系统间通通常使用的信息协议、网络 I/O 模型，以及网络 I/O 模型如何影响系统间通信的性能。接着讨论一款典型的 RPC 框架——Apache Thrift，包括它的使用方法和工作原理，来引导读者如何在系统间通信环境下找到性能、维护性、易用性之间的平衡。接着本部分内容将专门对系统间调用的异常问题进行描述，本书将向读者阐述为什么分布式事务不适合应用在高并发场景的移动互联网业务场景下，以及如何使用事务补偿机制来保证各业务系统上数据的一致性，最后简单讨论基于 Hystrix 的熔断器应用在系统间调用场景下的好处。

消息队列也是这部分内容的重点，系统间 RPC 完成的系统间调用和使用消息队列完成的调用，应用在两种完全不同的业务场景下，并且都可以达到系统间调用的最终业务要求。我们将讨论两款消息队列组件 Apache ActiveMQ 和 Apache Kafka 的基本使用和工作原理。

# 第 7 章

## 系统间通信：网络 I/O 模型

### 7.1 模型

首先我们来看一个现实场景：有两个技术人员 A 和 B，在进行一问一答形式的交流，如图 7-1 所示。



图 7-1 最简单的通信场景

下面来看这幅图中的几个要点：

- 图中两个对话的角色都使用中文进行交流。如果他们一人使用的是南斯拉夫语，而另一人使用的是索马里语，并且相互都不理解对方的语系，那么很显然 A 所要表达的语意就不能被 B 所理解。
- 他们的声音在空气中进行传播。空气除了支撑他们的呼吸，还支撑了他们声音的传播。如果没有空气他们是无法知道对方用中文说了什么的，甚至不能存活。
- 图中的场景可以看成角色 A 和角色 B 在进行第一次对话，对话的目的是为了协调他们



之间的交流方式：即角色 A 问完一个问题后需要等待角色 B 进行回答，收到角色 B 的回答后角色 A 才能问下一个问题。协调一致后，两个角色才能进行正式内容的交换。

- 由于都是人类，所以他们处理信息的方式也是一样的：用嘴说话，用耳朵听话，用大脑处理形成结果。虽然这两个角色都了解对方处理信息的方式，但是很显然他们并不关心这个事情：即便是现在突然来了一个外星人，只要他们使用的语言、交流方式协调一致了，就可以进行信息交流。
- 目前这个交流场景下，虽然只有角色 A 和角色 B 两个人。但是随时有可能增加第  $N$  个人，第  $N$  个人可能并不知道当前的信息交互采用哪种语言和交流方式。那么这里就留下来一个隐患，当新的角色 C 加入进来参与信息交互时，由哪个角色负责对 C 进行协调呢？

### 7.1.1 信息格式

很明显，通过中文进行交谈，两个人相互明白了对方的意图。为了保证信息传递的高效性，我们一定会将信息做成某种参与者都理解的格式。例如：中文有其特定的语法结构：“主谓宾，定状补”。在计算机系统通信过程中，为了保证信息能够被处理，信息也会被做成特定的格式，而且要确保处理器能够明白这种格式。常用的信息格式如下。

- XML：可扩展标记语言，这个语言格式由 W3C（万维网联盟）进行发布和维护。XML 语言格式应用广泛、扩展丰富，适合做网络通信的信息描述格式（一般是“应用层”协议）。例如 Google 定义的 XMPP 通信协议就是使用 XML 进行描述的；不过 XML 更广泛的使用场景是对系统环境进行描述的（因为它会造成较多的不必要的内容传输），例如服务器的配置描述、Spring 的配置描述、Maven 仓库描述，等等（当然以上举例中，还有更好的配置方式）。
- JSON：JSON（JavaScript Object Notation）是一种轻量级的数据交换格式。它和 XML 的设计思路是一致的——数据和语言无关性（流行的语言都支持 JSON 格式描述：Go、Python、C、C++、C#、Java、Erlang、JavaScript 等）；但是和 XML 不同，JSON 的设计目标就是为了进行通信。要描述同样的数据，JSON 格式的容量会更小。
- Protocol Buffer（PB）：Protocol Buffer 是 Google 的一种数据交换格式，它同样独立于语言、独立于平台。Google 提供了三种语言的实现：Java、C++ 和 Python，每一种实现都包含了相应语言的编译器以及库文件。
- TLV（三元组编码）：T（标记/类型域）L（长度/大小域）V（值/内容域），通常这种信息格式用于金融、军事领域。它通过字节的位运算来进行信息的序列化/反序列化（图 7-2）。

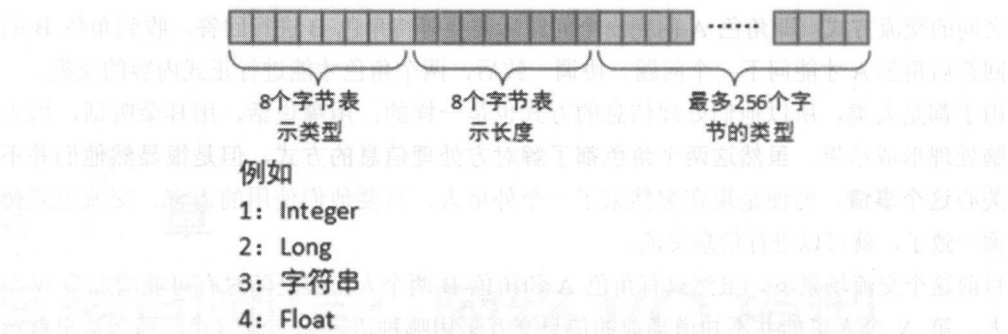


图 7-2 TLV 格式概要

- 自定义的格式：如果两个或多个内部系统已经约定好了一种信息格式，就可以使用自己定制的格式进行描述。可以使用 C++描述一个结构体，然后序列化/反序列化它，或者使用一个纯文本，以“|”号分割这些字符串，然后序列化/反序列化它。后文我们将介绍 Apache Thrift 的基本使用和序列化原理，读者可以观察到 Apache Thrift 对序列化后 byte 结构的组织方式就是一种自定义的方式。

7.1.2 网络协议

如 7.1 节图 7-1 所描述的场景，有一个我们看不到但是却很重要的元素：空气。声音只有在空气中才能完成传播，因为真空无法传播声音。同样，两个计算机间的信息通信是在网络中完成传播的，没有网络就没法传播信息。网络协议就是计算机领域的“空气”，图 7-3 中我们以 OSI 模型作为参考。

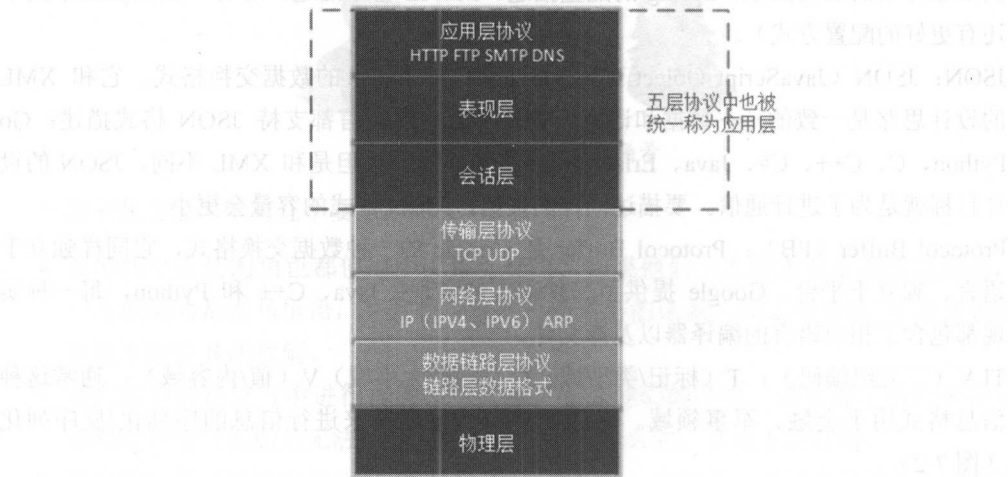


图 7-3 七层/五层 网络协议结构

- 物理层：物理层就是网络设备层，例如网卡、交换机等设备，在它们之间一般传递的是电信号或者光信号。
- 数据链路层：数据链路又分为物理链路和逻辑链路。物理链路负责组合一组电信号，称之为“帧”；逻辑链路通过一些规则和协议保证帧传输的正确性，并且可以使来自于多个源（目标）的帧内容在同一个物理链路上进行传输，实现“链路复用”。
- 网络层：网络层使用最广泛的协议是 IP 协议（又分为 IPV4 协议和 IPV6 协议）、IPX 协议。这些协议解决的是源和目标的定位问题，以及从源如何到达目标的问题。
- 传输层：TCP、UDP 是传输层最常使用的协议，传输层的最重要工作就是携带内容信息，并且通过它们的协议规范提供某种通信机制。举例来说，TCP 协议中的通信机制是：首先进行三次通信握手，然后再进行正式数据的传送，并且通过校验机制保证每个数据报文的正确性，如果数据报文错误了，则重新发送。
- 应用层：包括 HTTP 协议、FTP 协议、TELNET 协议在内的众多协议都是应用层协议。应用层协议是最灵活的协议，甚至可以由程序员自行定义应用层协议。如图 7-4 所示表示了 HTTP 协议的工作步骤。

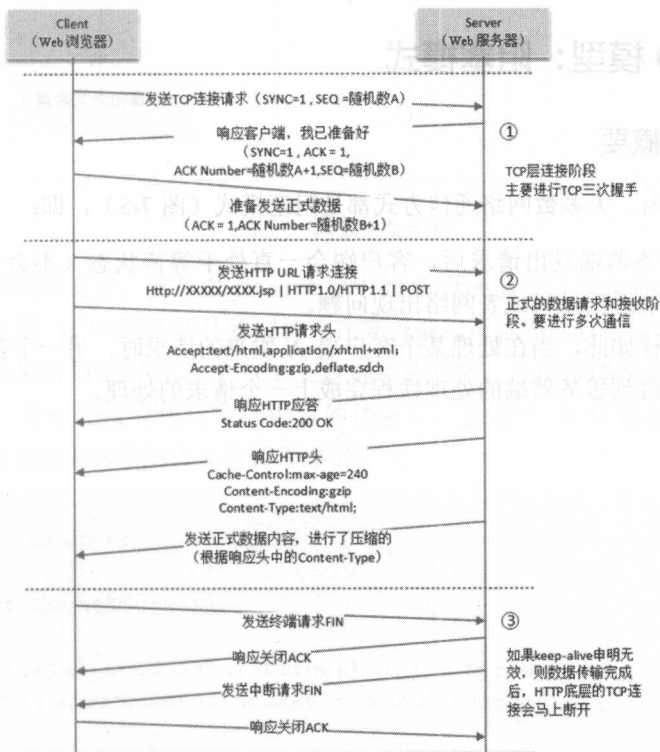


图 7-4 HTTP 协议通信过程概要

本书中，不会把网络协议作为一个重点。这是因为网络协议的知识是一个相对独立的知识领域。如果读者对网络协议有兴趣，这里推荐两本书：《TCP/IP 详解，卷 1：协议》和《TCP/IP 详解，卷 2：实现》。

### 7.1.3 通信方式/框架

在 7.1 节开头的对话场景中，我们看到其中一个角色在对话开始之初就规定了一种沟通方式：“你必须把我说的话听完，然后给我反馈后，我才会问第二个问题”。这种沟通方式虽然沟通效率不高，但是很有效：一个问题一个问题地处理。

但是如果参与沟通的人处理信息的能力比较强，那么他们还可以采用另一种沟通方式：“我给自己提的问题编了一个号，在问完第  $X$  个问题后，我不会等待你返回，就会问第  $X+1$  个问题，同样你在听完我第  $X$  个问题后，一边处理我的问题，一边听我第  $X+1$  个问题。”

实际上以上两种现实中的沟通方式，在计算机领域是可以找到对应的通信方式的，这就是本书马上会着重讨论的目前使用的四种网络 I/O 模型。

## 7.2 网络 I/O 模型：阻塞模式

### 7.2.1 通信模型概要

很长一段时间内，大多数网络通信方式都是阻塞模式（图 7-5），即：

- 客户端向服务器端发出请求后，客户端会一直处于等待状态（不会再做其他事情），直到服务器端返回结果或者网络出现问题。
- 服务器端同样如此，当在处理某个客户端 A 发来的请求时，另一个客户端 B 发来的请求会等待，直到服务器端的处理线程完成上一个请求的处理。

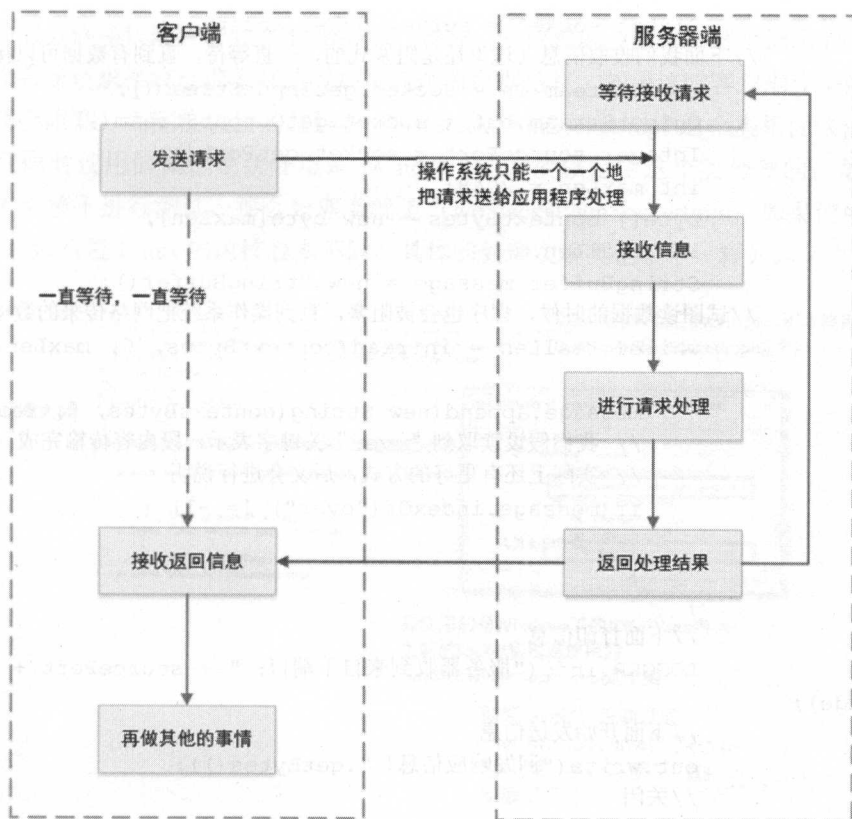


图 7-5 阻塞模型示例

Java 对阻塞模式的支持，就是由 `java.net` 包中的 `Socket` 套接字功能完成的。这里要说明一下，`Socket` 套接字是 TCP/UDP 等传输层协议在高级编程语言中的具体体现。例如客户端使用 TCP 协议连接这台服务器的时候，当 TCP 三次握手成功后，应用程序就会创建一个 `Socket` 套接字对象（注意，这时还没有进行数据内容的传输），当这个 TCP 连接出现数据传输时，`Socket` 套接字就会把数据传输的表现告诉程序员。

```

.....
package testBSocket;
.....
public class SocketServer1 {
    .....
    public static void main(String[] args) throws Exception{
        ServerSocket serverSocket = new ServerSocket(83);
        try {
            while(true) {
                //这里 Java 通过 JNI 请求操作系统，并等待操作系统返回结果或出错
            }
        }
    }
}
  
```

```

        Socket socket = serverSocket.accept();
        //下面我们收取信息（这里还是阻塞式的，一直等待，直到有数据可以接收）
        InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream();
        Integer sourcePort = socket.getPort();
        int maxLen = 2048;
        byte[] contextBytes = new byte[maxLen];
        int realLen;
        StringBuffer message = new StringBuffer();
        //试图读数据的时候，程序也会被阻塞，直到操作系统把网络传来的数据准备好。
        while((realLen = in.read(contextBytes, 0, maxLen)) != -1)
        {
            message.append(new String(contextBytes, 0, realLen));
            // 我们假设读取到“over”关键字表示一段内容传输完成
            // 实际上还有更好的方式，后文会进行说明
            if(message.indexOf("over") != -1) {
                break;
            }
        }
        //下面打印信息
        LOGGER.info("服务器收到来自于端口：" + sourcePort + "的信息："
            + message);

        //下面开始发送信息
        out.write("回发响应信息!".getBytes());
        //关闭
        out.close();
        in.close();
        socket.close();
    }
    catch(Exception e) {
        SocketServer1.LOGGER.error(e.getMessage(), e);
    }
    finally {
        if(serverSocket != null) {
            serverSocket.close();
        }
    }
}
}
.....

```

请注意以上示例代码中的关于传输结束标记“over”的说明，这实际上是一个关于“粘包”和“半包”的问题。在实际工作场景下我们会有更好的方式来处理这个问题，这个问题怎么处理要到本书后续的章节中才会讨论到。所以读者目前只需要认为这是“信息从客户端读取完毕”的标记即可，并将注意力集中在本节讲解的网络 I/O 模型上。



上面的服务器端代码执行到 `serverSocket.accept()` 的位置就会阻塞，这个调用的含义是应用程序向操作系统请求接收已准备好的客户端连接的数据信息，如果这时客户端连接还没有准备好，代码就会阻塞，而底层调用的位置在 `DualStackPlainSocketImpl` 这个类里面（注意笔者在运行这段代码时使用的操作系统环境是 Windows 8，所以是由这个类处理的；如果读者在 Windows 7 环境下进行测试，那么处理类就是 `TwoStacksPlainSocketImpl`；如果使用的测试环境是 Linux，那么视 Linux 的内核版本不同，具体的处理实现类也就不一样）。

这是大师

```

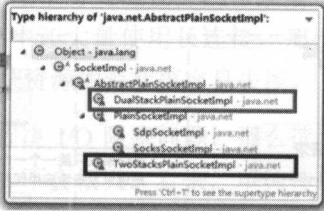
/**
 * This class defines the plain SocketImpl that is used on Windows platforms
 * greater or equal to Windows Vista. These platforms have a dual
 * layer TCP/IP stack and can handle both IPv4 and IPV6 through a
 * single file descriptor.
 *
 * @author Chris Hegarty
 */
class DualStackPlainSocketImpl extends AbstractPlainSocketImpl {
    static javaIOFileDescriptorAccess fdAccess = SharedSec...

    // true if this socket is exclusively bound
    private final boolean exclusiveBind;

    // emulates SO_REUSEADDR when exclusiveBind is true
    private boolean isReuseAddress;

```

Vista内核和之后的Windows版本使用  
`DualStackPlainSocketImpl`这个类



我使用的是Windows下的JDK，Vista内核  
之前的Socket实现是使用的  
`TwoStacksPlainSocketImpl`这个类

如果Java程序没有设置  
timeout，那么Java程序在调  
用JNI时，会一直等待，直到  
有数据返回

```

130     if (timeout <= 0) {
131         newfd = accept0(nativefd, isaa);
132     } else {
133         configureBlocking(nativefd, false);
134         try {
135             waitforNewConnection(nativefd, timeout);
136             newfd = accept0(nativefd, isaa);
137             if (newfd != -1) {
138                 configureBlocking(newfd, true);
139             }
140         } finally {
141             configureBlocking(nativefd, true);
142         }
143     }

```

图 7-6 阻塞模型工作类

传统的阻塞模型在通信方式上存在几个问题：

- 同一时间，服务器只能接收来自于客户端 A 的请求信息；虽然客户端 A 和客户端 B 的请求是同时进行的，但客户端 B 发送的请求信息也只能等到服务器接收完客户端 A 的请求数据后，才能被接收。
- 由于服务器一次只能处理一个客户端请求，当处理完成并返回后（或者异常时），才能进行第二次请求的处理。很显然，这样的处理方式在高并发的情况下，是不建议采

用的。

上面说的是服务器只有一个线程的情况，也许有的读者会直接提出可以使用多线程技术来解决这个问题（图 7-7）：

- 当服务器收到客户端 X 的请求后（读取到所有请求数据后），将这个请求送入一个独立线程进行处理，然后主线程继续接收客户端 Y 的请求。
- 客户端一侧也可以使用一个子线程和服务端进行通信。这样客户端主线程的其他工作就不受影响了，当服务器端有响应信息时再由这个子线程通过监听模式/观察模式或者类似的其他设计模式通知主线程。

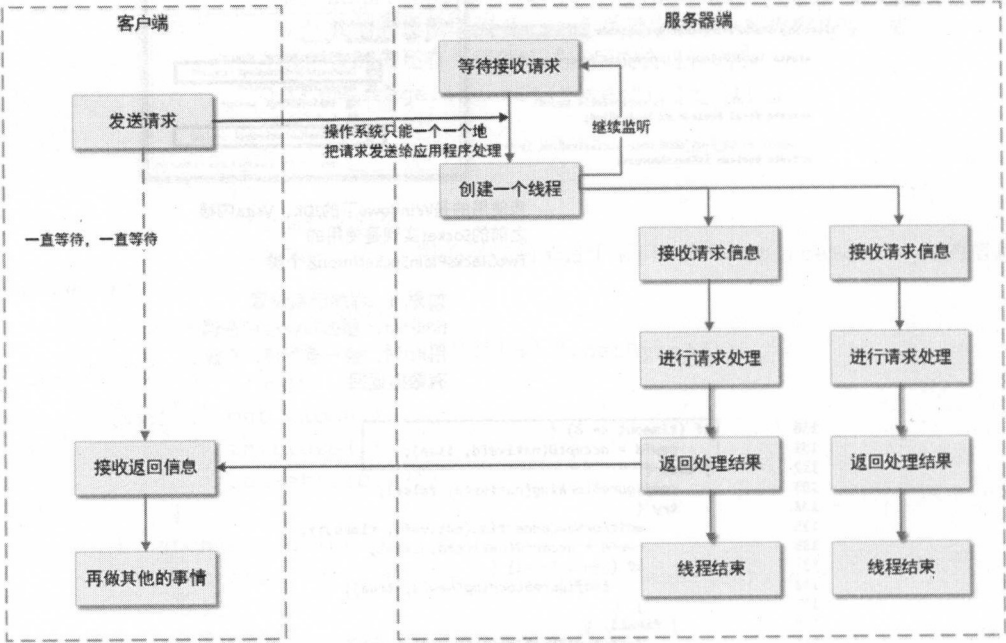


图 7-7 阻塞模式的多线程优化

但是使用多线程来解决这个问题实际上是有局限性的：

- 虽然在服务器端，接收到数据后的处理交给了一个独立线程进行，但是操作系统通知 `accept()` 的方式还是单个线程运行的。也就是说，实际上是服务器接收到数据报文后的“业务处理过程”可以应用多线程技术，但是数据报文的接收还是需要一个接一个地来，从以上的示例代码和其调试过程我们都可以明确看到这一点。
- 在 Linux 系统中，可以创建的线程是有限的。可以通过 `cat /proc/sys/kernel/threads-max` 命令查看可以创建的最大线程数。当然这个值可以更改，但是参与调度的线程数量越

大，CPU 用在线程间切换所需的时间也就越长，用来处理真正业务的资源也就越少。**CPU 线程状态间切换的性能消耗是非常巨大的**，后文我们会对这个描述给出相应的实例以便将这个感性的认识具体化，帮助读者真正认识线程间状态切换的性能代价。

- 创建一个线程是有较大的资源消耗的。例如 JVM 创建一个线程时，即使这个线程不做任何工作，JVM 都会分配一个独立线程栈空间（不同 JDK 版本默认的大小不一样）。虽然它可以通过“-Xss”参数进行大小调整，但这不影响 CPU 一级、二次缓存中的数据出现线程数据的换入/换出。
- 当然，还可以使用类似 Java 语言中 ThreadPoolExecutor 这样的线程池来缓解线程创建和切换的资源消耗问题，但是又会造成线程池中待处理任务的持续增加，同样消耗了大量内存资源。另外，如果读者的应用程序中还大量使用长连接，那么线程池中的线程就会被对应的任务持续占用，这样一来系统资源的消耗更容易失控。

所以，如果你真想单纯使用多线程的方法来解决 I/O 阻塞问题，那么读者自己都可以感觉到一个服务器节点可以同时接收数据的并发能力并不会很大。看来，单纯使用线程解决这个问题不是最好的办法。

## 7.2.2 阻塞模式深入分析

实际上，通过 7.2.1 节对阻塞模型的介绍我们就可以分析出，阻塞模型的问题关键不在于是否使用了多线程（包括线程池）处理并发请求，而在于 **accept()、read()** 的操作点都被阻塞了。要近似模拟这个问题也很简单：我们模拟了 20 个客户端（用 20 根线程模拟），利用 Java 的同步计数器 CountDownLatch 保证这 20 个客户都初始化完成后再同时向服务器发送请求，再来观察一下 Server 侧接收信息的情况。

- 客户端代码（SocketClientDaemon）

```
package testBSocket;
import java.util.concurrent.CountDownLatch;
public class SocketClientDaemon {
    public static void main(String[] args) throws Exception {
        Integer clientNumber = 20;
        CountDownLatch countDownLatch = new CountDownLatch(clientNumber);
        //分别开始启动这 20 个客户端
        for(int index = 0 ; index < clientNumber ; index++ , countDownLatch.countDown()) {
            SocketClientRequestThread client = new SocketClientRequestThread(countDownLatch, index);
            new Thread(client).start();
        }
        //这个同步锁不涉及具体的实验逻辑，只是保证守护线程在启动所有线程后，不会退出
```

```

        synchronized (SocketClientDaemon.class) {
            SocketClientDaemon.class.wait();
        }
    }
}

```

### • 客户端代码 (SocketClientRequestThread 模拟请求)

```

package testBSocket;

.....
// 一个 SocketClientRequestThread 线程模拟一个客户端请求
public class SocketClientRequestThread implements Runnable {
    private static final Log LOGGER = LoggerFactory.getLog(SocketClientRequestThread.class);
    private CountdownLatch countDownLatch;
    // 这个线程的编号
    private Integer clientIndex;
    // countDownLatch 是 Java 提供的线程同步计数器。
    // 当计数器数值减为 0 时，所有受其影响而阻塞的线程将会被激活。尽可能模拟并发请求的
    // 真实性（但实际上也并不是完全并发的）
    public SocketClientRequestThread(CountDownLatch countDownLatch,
    Integer clientIndex) {
        this.countDownLatch = countDownLatch;
        this.clientIndex = clientIndex;
    }
    @Override
    public void run() {
        Socket socket = null;
        OutputStream clientRequest = null;
        InputStream clientResponse = null;
        try {
            socket = new Socket("localhost", 83);
            clientRequest = socket.getOutputStream();
            clientResponse = socket.getInputStream();
            // 阻塞，直到 SocketClientDaemon 完成所有线程的启动，然后所有线程一起发送请求
            this.countDownLatch.await();
            // 发送请求信息
            clientRequest.write(("这是第" + this.clientIndex + " 个客户端的
            请求。").getBytes());
            clientRequest.flush();
            // 在这里等待，直到服务器返回信息
            SocketClientRequestThread.LOGGER.info("第" + this.clientIndex
            + " 个客户端的请求发送完成，等待服务器返回信息");
            int maxLen = 1024;
            byte[] contextBytes = new byte[maxLen];

```



```

        int realLen;
        String message = "";
        // 程序执行到这里，会一直等待服务器返回信息
        // （注意，前提是 in 和 out 都不能关闭，如果关闭了就收不到）
        while((realLen = clientResponse.read(contextBytes, 0,
maxLen)) != -1) {
            message += new String(contextBytes, 0, realLen);
        }
        SocketClientRequestThread.LOGGER.info("接收到来自服务器的信息："
+ message);
    } catch (Exception e) {
        SocketClientRequestThread.LOGGER.error(e.getMessage(), e);
    } finally {
        // 记得关闭连接
        .....
    }
}
}

```

- 服务器端 (SocketServer1) 单个线程

同 7.2.1 节中服务器端的代码示例。

通过运行和单步调试以上客户端代码和服务器端代码，我们可以发现，即使保证了 20 个客户端请求同时发送给服务器端，服务器端通过阻塞模型实现的处理代码也只能一次处理一个客户端请求，其他的请求必须依次等待处理完成。

就像上文所述我们可以使用多线程来优化服务器端的处理过程，客户端代码和本节一致，最主要是更改服务器端的代码：

```

package testBSocket;
.....
public class SocketServer2 {
    private static final Log LOGGER = LogFactory.getLog(SocketServer2.
class);
    public static void main(String[] args) throws Exception{
        ServerSocket serverSocket = new ServerSocket(83);
        try {
            while(true) {
                Socket socket = serverSocket.accept();
                //业务处理过程可以交给一个线程，不过线程的创建很耗资源
                //最终改变不了，accept() 只能在被阻塞的情况一个一个接收 Socket
                SocketServerThread socketServerThread = new SocketServer
Thread(socket);
                new Thread(socketServerThread).start();
            }
        }
    }
}

```

```

    }
    } catch (Exception e) {
        SocketServer2.LOGGER.error(e.getMessage(), e);
    } finally {
        if (serverSocket != null) {
            serverSocket.close();
        }
    }
}

//当然, 接收到客户端的 Socket 后, 业务的处理过程可以交给一个线程来做
class SocketServerThread implements Runnable {
    private static final Log LOGGER = LogFactory.getLog(SocketServerThread.class);

    private Socket socket;
    public SocketServerThread(Socket socket) {
        this.socket = socket;
    }
    @Override
    public void run() {
        InputStream in = null;
        OutputStream out = null;
        try {
            //下面我们收取信息
            in = socket.getInputStream();
            out = socket.getOutputStream();
            Integer sourcePort = socket.getPort();
            int maxLen = 1024;
            byte[] contextBytes = new byte[maxLen];
            //同样无法改变 read 方法被阻塞, 直到操作系统有数据准备好的现象
            int realLen = in.read(contextBytes, 0, maxLen);
            String message = new String(contextBytes, 0, realLen);
            LOGGER.info("服务器收到来自于端口: " + sourcePort + "的信息: " +
message);

            //下面开始发送响应信息
            out.write("回发响应信息!".getBytes());
        } catch (Exception e) {
            SocketServerThread.LOGGER.error(e.getMessage(), e);
        } finally {
            // 试图关闭连接
            .....
        }
    }
}

```



阅读到这里，笔者相信服务器使用单线程的效果就不用再介绍了，我们主要看一看服务器使用多线程处理时的效果（图 7-8）。

客户端20个请求同时发送完成后，都等待服务器端返回信息

即使服务器端使用了线程来处理请求，但是也避免不了上文中提到的问题

```

21 public static void main(String[] args) throws Exception{
22     ServerSocket serverSocket = new ServerSocket(83);
23
24     try {
25         while(true) {
26             Socket socket = serverSocket.accept();
27             //当然业务处理过程可以交给一个线程（这里可以使用线程池），并且线程的创建是很耗资源的，
28             //最终改变不了.accept()只能一个一个接受socket的情况
29             SocketServerThread socketServerThread = new SocketServerThread(socket);
30             new Thread(socketServerThread).start();
31         }
32     } catch (Exception e) {
33         SocketServer2.LOGGER.error(e.getMessage(), e);
34     } finally {
35         if(serverSocket != null) {
36             serverSocket.close();
37         }
38     }
39 }
40
41
42 /**
43  * 当然，接收来自客户端的socket后，业务的外理过程可以交给一个线程来做。

```

图 7-8 多线程执行效果

### 7.2.3 问题的根源

问题的关键并不是“是否使用了多线程”，而是为什么 `accept()`、`read()` 方法会被阻塞，本节就对这个问题分析。另外说明一下，本章 7.4 节将要介绍的多路复用 I/O 模型就是为了解决这样的并发性问题而出现的，但为了后面说明多路复用 I/O 模型做好铺垫，本书有必要首先说清楚同步阻塞式模型、同步非阻塞式模型、多路复用模型之间的联系和区别。

API 文档中对于 `serverSocket.accept()` 方法的使用描述如下：

Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

那么我们来看看为什么 `serverSocket.accept()` 会被阻塞。这里涉及阻塞式同步 I/O 的工作原理：

- 服务器线程发起一个 `accept()` 动作，询问操作系统是否有新的 Socket 套接字信息从端口 X 发送过来，并且准备好（图 7-9）。

如果Java程序没有设置 timeout，那么Java程序在调用JNI时，会一直等待，直到有数据返回

```
130         if (timeout <= 0) {
131             newfd = accept0(nativefd, isaa);
132         } else {
133             configureBlocking(nativefd, false);
134             try {
135                 waitForNewConnection(nativefd, timeout);
136                 newfd = accept0(nativefd, isaa);
137                 if (newfd != -1) {
138                     configureBlocking(newfd, true);
139                 }
140             } finally {
141                 configureBlocking(nativefd, true);
142             }
143         }
144     }
```

图 7-9 最简单的阻塞模型支持

- 注意，是询问操作系统。也就是说 Socket 套接字的网络 I/O 模型支持是基于操作系统的，实际上各种网络 I/O 模型都需要操作系统级别的支持，如图 7-10 所示。

```

static native void localAddress(int fd, InetAddressContainer in) throws SocketException;
static native void listen0(int fd, int backlog) throws IOException;
static native int accept0(int fd, InetSocketAddress[] isaa) throws IOException;
static native void waitForNewConnection(int fd, int timeout) throws IOException;
static native int available0(int fd) throws IOException;
static native void close0(int fd) throws IOException;
static native void shutdown0(int fd, int howto) throws IOException;
static native void setIntOption(int fd, int cmd, int optionValue) throws SocketException;
static native int getIntOption(int fd, int cmd) throws SocketException;
static native void sendOOB(int fd, int data) throws IOException;
static native void configureBlocking(int fd, boolean blocking) throws IOException;

```

图 7-10 JNI BIO 内部方法

如果操作系统没有从指定的端口 X 接收到数据，那么操作系统就会等待。这样 `serverSocket.accept()` 方法也就会一直等待！这就是为什么 `accept()` 方法会阻塞：它内部的实现是使用操作系统级别的同步网络 I/O 模型。

**阻塞式 I/O 和非阻塞式 I/O：**这两个概念是程序级别的。主要描述的是程序请求操作系统 I/O 操作后，如果网络 I/O 资源没有准备好，那么程序该如何处理的问题：前者等待，后者继续执行（并且使用线程一直轮询，直到有 I/O 资源准备好了）。

**同步 I/O 和非同步 I/O：**这两个概念是操作系统级别的。主要描述的是操作系统在收到程序请求网络 I/O 操作后，如果网络 I/O 资源没有准备好，该如何响应程序的问题：前者不响应，直到网络 I/O 资源准备好；后者返回一个标记（好让程序和自己知道以后的数据往哪里通知），当网络 I/O 资源准备好以后，再用事件机制返回给程序。

### 7.3 网络 I/O 模型：同步非阻塞模式——对阻塞模式的改进

对于阻塞方式的一种改进是在应用程序层面上将“一直等待”的状态主动打开，如图 7-11 所示。

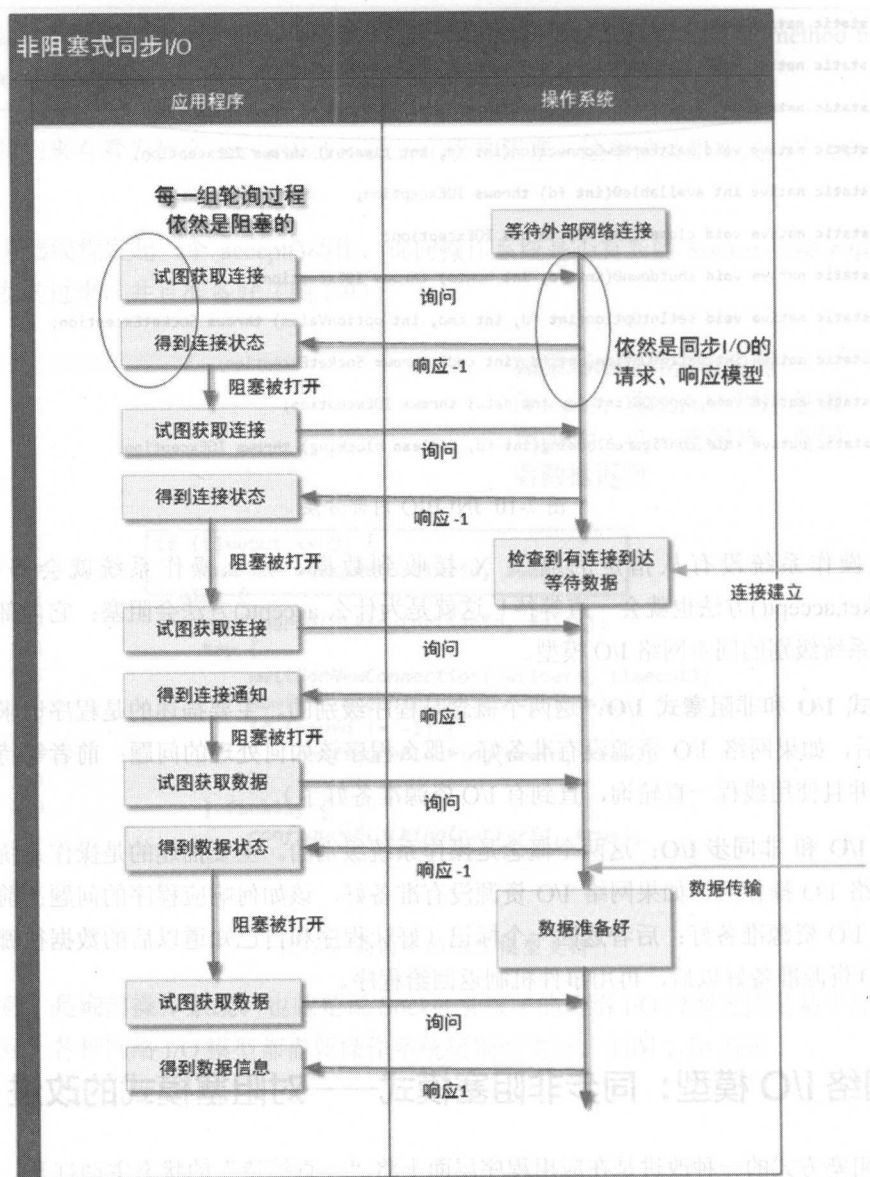


图 7-11 对阻塞模型的改进

这种模式下，应用程序的线程不再一直等待操作系统的 I/O 状态，而是在等待一段时间后就解除阻塞。如果没有得到想要的结果，则再次进行相同的操作。这样的工作方式，保证了应用程序的线程不会一直阻塞，而可以进行一些其他工作——例如软件业务层面上暂时不需要这些网络数据的操作过程（图 7-12）。

如果Java程序没有设置 timeout, 那么Java程序在调用JNI时, 会一直等待, 直到有数据返回

```

130     if (timeout <= 0) {
131         newfd = accept0(nativefd, isaa);
132     } else {
133         configureBlocking(nativefd, false);
134         try {
135             waitForNewConnection(nativefd, timeout);
136             newfd = accept0(nativefd, isaa);
137             if (newfd != -1) {
138                 configureBlocking(newfd, true);
139             }
140         } finally {
141             configureBlocking(nativefd, true);
142         }
143     }

```

如果设置了timeout, 那么Java程序会设置JNI, 只等待timeout的时间, 然后返回。一定注意newfd == -1的情况, 表示这次accept没有发现有数据从底层返回。

图 7-12 Java 对阻塞模式的处理改进

那么 timeout 是在哪里被设置的呢? 可以在 ServerSocket 中, 调用 DualStackPlainSocketImpl 的父类 SocketImpl, 从中找到 timeout 的设置位置, 如图 7-13 所示。

```

634 public synchronized void setSoTimeout(int timeout) throws SocketException {
635     if (isClosed())
636         throw new SocketException("Socket is closed");
637     getImpl().setOption(SocketOptions.SO_TIMEOUT, new Integer(timeout));
638 }

```

图 7-13 timeout 的设置位置

ServerSocket 中的 setSoTimeout 方法也有相应的注释说明:

Enable/disable SO\_TIMEOUT with the specified timeout, in milliseconds. With this option set to a non-zero timeout, a call to accept() for this ServerSocket will block for only this amount of time. If the timeout expires, **a java.Net.SocketTimeoutException is raised, though the ServerSocket is still valid.** The option must be enabled prior to entering the blocking operation to have effect. The timeout must be > 0. A timeout of zero is interpreted as an infinite timeout.

以下代码片段展示了一个 Java 对阻塞模式的设置改变:

```

package testBSocket;

.....

public class SocketServer2 {
    private static Object xWait = new Object();
    .....
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(83);
            serverSocket.setSoTimeout(100);
            while(true) {
                Socket socket = null;
                try {
                    // 程序不会一直在这里阻塞了
                    socket = serverSocket.accept();
                } catch(SocketTimeoutException e1) {
                    //=====
                    // 执行到这里,说明本次 accept()方法没有接收到任何数据报文
                    // 主线程在这里就可以做一些事情,记为 x
                    //=====
                    synchronized (SocketServer2.xWait) {
                        LOGGER.info("这次没有接收到 TCP 连接,据报文,等待 10 毫
秒,模拟事件 x 的处理时间");
                        SocketServer2.xWait.wait(10);
                    }
                    continue;
                }
                InputStream in = socket.getInputStream();
                OutputStream out = socket.getOutputStream();
                Integer sourcePort = socket.getPort();
                int maxLen = 2048;
                byte[] contextBytes = new byte[maxLen];
                int realLen;
                StringBuffer message = new StringBuffer();
                // 以下接收数据,与 7.2.1 节中的代码处理过程一致
                .....
            }
        } catch(Exception e) {
            SocketServer2.LOGGER.error(e.getMessage(), e);
        } finally {
            // 关闭连接
            .....
        }
    }
}

```



```
}

```

以上代码片段的执行结果如图 7-14 所示。



```

29
30 try {
31     serverSocket = new ServerSocket(83);
32     serverSocket.setSoTimeout(100);
33     while(true) {
34         Socket socket = null;
35         try {
36             socket = serverSocket.accept();
37         } catch(SocketTimeoutException e1) {
38             // 执行到这里，说明本次accept没有接收到任何数据报文
39             // 主线程在这里就可以做一些事情，记为X
40             // 模拟事件X的处理时间
41             synchronized (SocketServer2.xWait) {
42                 SocketServer2.LOGGER.info("这次没有从底层接收到任务数据报文，等待10毫秒，模拟事件X的处理时间");
43                 SocketServer2.xWait.wait(10);
44             }
45         }
46     }
47 }

```

Console [Java Application] D:\Program Files\Java\jre7\bin\javaw.exe (2015年9月16日 下午5:27:36)

```

0 [main] INFO test8Socket.SocketServer2 - 这次没有从底层接收到任务数据报文，等待10毫秒，模拟事件X的处理时间
111 [main] INFO test8Socket.SocketServer2 - 这次没有从底层接收到任务数据报文，等待10毫秒，模拟事件X的处理时间
221 [main] INFO test8Socket.SocketServer2 - 这次没有从底层接收到任务数据报文，等待10毫秒，模拟事件X的处理时间
331 [main] INFO test8Socket.SocketServer2 - 这次没有从底层接收到任务数据报文，等待10毫秒，模拟事件X的处理时间
441 [main] INFO test8Socket.SocketServer2 - 这次没有从底层接收到任务数据报文，等待10毫秒，模拟事件X的处理时间

```

图 7-14 非阻塞模式第一次编码的执行效果

这里我们针对 SocketServer 增加了阻塞等待时间，实际上只实现了非阻塞 I/O 模型中的第一步：监听连接状态的非阻塞。通过运行代码，我们可以发现接收数据的 `read()` 方法还是被阻塞的。

### 7.3.1 首次改进

那么我们能不能改进 `read()` 方式，让它也不一直被阻塞呢？当然是可以的，Socket 套接字同样支持等待超时设置。代码如下：

```

.....
public class SocketServer3 {
    private static Object xWait = new Object();
    .....
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(83);
            serverSocket.setSoTimeout(100);
            while(true) {
                Socket socket = null;
                try {
                    socket = serverSocket.accept();

```

```

    } catch(SocketTimeoutException e1) {
        //=====
        // 执行到这里,说明本次 accept()方法没有接收到任何 TCP 连接
        // 主线程在这里就可以做一些事情,记为 X
        //=====
        synchronized (SocketServer3.xWait) {
            LOGGER.info("这次没有接收到 TCP 连接,等待 10 毫秒,模拟事件
            X 的处理时间");
            SocketServer3.xWait.wait(10);
        }
        continue;
    }
    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();
    Integer sourcePort = socket.getPort();
    int maxLen = 2048;
    byte[] contextBytes = new byte[maxLen];
    int realLen;
    StringBuffer message = new StringBuffer();
    //下面我们收取信息(非阻塞方式,read()方法的等待超时时间)
    socket.setSoTimeout(10);
    BIORead:while(true) {
        try {
            while((realLen = in.read(contextBytes, 0,
maxLen)) != -1) {
                message.append(new String(contextBytes, 0,
realLen));

                // 我们同样假设读取到“over”关键字,表示业务内容传输完成
                if(message.indexOf("over") != -1) {
                    break BIORead;
                }
            }
        } catch(SocketTimeoutException e2) {
            //=====
            // 执行到这里,说明本次 read()方法没有接收到任何数据流
            // 主线程在这里又可以做一些事情,记为 Y
            //=====
            LOGGER.info("这次没有接收到任务数据报文,等待 10 毫秒,
模拟事件 Y 的处理时间");

            continue;
        }
    }
    //下面打印信息
    LOGGER.info("服务器收到来自于端口: " + sourcePort + "的信息:

```

```

" + message);

        //下面开始发送信息
        out.write("回发响应信息!".getBytes());
        //关闭 in 和 out 对象
        .....
    }
} catch(Exception e) {
    LOGGER.error(e.getMessage(), e);
} finally {
    // 关闭服务
    .....
}
.....

```

这样一来，我们完成了对阻塞模型的第一次改进：让 TCP 连接和数据读取这两个过程，都变成了“非阻塞”方式。

当然，这种方式对网络 I/O 性能的提升意义不大，原因是这种处理方式实际上并没有解决 `accept()` 方法、`read()` 方法阻塞的根本问题。根据上文的描述，`accept()` 方法、`read()` 方法阻塞的根本问题是底层接收数据时采用了操作系统提供的“同步 I/O”工作方式。这两次改进过程，只是解决了 I/O 操作的两步中的第一步：将程序层面的阻塞方式变成了非阻塞方式。

### 7.3.2 再次改进

另外，由于应用程序级别并没有使用多线程技术，这就导致了应用程序只能一个一个地对 Socket 套接字进行处理。这个 Socket 套接字没有处理完，就没法处理下一个 Socket 套接字。针对这个问题还是可以进行改进的：让应用程序层面上各个 Socket 套接字的处理相互不影响。以下代码片段是在上一次改进的基础上加入了多线程技术：

```

package testBSocket;

.....
//通过加入线程的概念，让 Socket Server 能够在应用层面
//通过非阻塞的方式同时处理多个 Socket 套接字
public class SocketServer4 {
    private static Object xWait = new Object();
    private static final Log LOGGER = LogFactory.getLog(SocketServer4.
class);

    public static void main(String[] args) throws Exception{
        ServerSocket serverSocket = new ServerSocket(83);
        serverSocket.setSoTimeout(100);
        try {
            while(true) {
                Socket socket = null;

```



```

        try {
            socket = serverSocket.accept();
        } catch(SocketTimeoutException e1) {
            // 和 7.3.1 节中的代码一致
        }
        // 业务处理过程可以交给一个线程（这里可以使用线程池）
        // 注意，线程的创建过程是很耗资源和时间的
        // 最终改变不了 accept() 方法只能一个一个地接收 Socket 连接的情况
        SocketServerThread socketServerThread = new SocketServer
Thread(socket);
        new Thread(socketServerThread).start();
    }
    } catch(Exception e) {
        LOGGER.error(e.getMessage(), e);
    } finally {
        if(serverSocket != null) {
            serverSocket.close();
        }
    }
}
}
//当然，接收到客户端的 Socket 后，业务的处理过程可以交给一个线程来做
class SocketServerThread implements Runnable {
    private static final Log LOGGER = LogFactory.getLog(SocketServerThread.
class);
    private Socket socket;
    public SocketServerThread (Socket socket) {
        this.socket = socket;
    }
    @Override
    public void run() {
        InputStream in = null;
        OutputStream out = null;
        try {
            in = socket.getInputStream();
            out = socket.getOutputStream();
            Integer sourcePort = socket.getPort();
            int maxLen = 2048;
            byte[] contextBytes = new byte[maxLen];
            int realLen;
            StringBuffer message = new StringBuffer();
            //下面我们收取信息
            this.socket.setSoTimeout(10);
            BIORead:while(true) {

```

```

        // 这个过程与 7.3.1 节中接收数据的过程基本一致
        .....
    }
    //下面打印信息
    Long threadId = Thread.currentThread().getId();
    SocketServerThread.LOGGER.info("服务器(线程: " + threadId + ")
收到来自于端口: " + sourcePort + "的信息: " + message);
    //下面开始发送信息
    out.write("回发响应信息!".getBytes());
    //关闭 in 和 out 对象
    .....
} catch(Exception e) {
    LOGGER.error(e.getMessage(), e);
}
}
}

```

### 7.3.3 依然存在问题

引入了多线程技术后，I/O 的处理吞吐量有了提高（实际上并不显著），但是这样做就真的没有问题了吗？显然还是有的，因为 `accept()` 方法为“同步”工作的情况依然存在。而使用多线程解决这个问题的局限性在 7.2.1 节的最后部分也有详细描述。

需要说明的是，7.3 节所讨论的问题中无论是 **Java** 对阻塞模型的支持，还是通过 **Java** 对阻塞模型改进后的非阻塞模型的支持，都不是我们俗称的 **Java NIO**，它们都还属于我们所说的 **BIO**（阻塞）模型，因为最核心的 `accept()` 方法的工作原理并没有得到根本性的改变，所谓非阻塞只是一个错觉罢了。

## 7.4 网络 I/O 模型：多路复用（I/O Multiplex）

我们试想一下这样的现实场景：

一个餐厅同时有 100 位客人到店，他们到店后要做的第一件事情就是点菜。但是问题来了，餐厅老板为了节约人力成本目前只有一位大堂服务员拿着唯一的一本菜单等待客人进行服务。

那么最粗暴（但是最简单）的方法是（记为方法 A）：无论有多少客人等待点餐，服务员都把仅有的一份菜单递给其中一位客人，然后站在客人身旁等待这个客人完成点菜过程。在记录客人点菜内容后，把点菜记录交给后堂厨师。然后再来到第二位客人面前将以上工作方式重复一次……然后是第  $X$  位客人。很明显，这样设置服务流程是不行的。因为随后的 80 位客人，在等待超时后就会离店（还会给差评哦）。

于是，餐厅老板通过一种办法（记为方法 B）进行了改进。老板立刻雇用 99 名服务员，同时印制 99 本新的菜单。每一名服务员手持一份菜单负责一位客人（关键不只在服务员，还在于菜单。因为没有菜单客人也无法点菜）。在客人点完菜后，记录点菜内容交给后堂厨师（当然为了更高效，后堂厨师最好也有 100 名）。这样每一位客人享受的都是 VIP 服务，客人当然不会走还会给好评。但是高昂的人力成本就让人头疼了。

另外一种办法（记为方法 C），就是改进点菜的方式：当客人到店后，自己领取一份菜单。想好自己要点的菜后，再呼叫服务员。服务员站在客人身边记录点菜内容。将菜单递给厨师的过程也要进行改进，并不是每一份菜单记录好以后，都要交给后堂厨师。服务员可以记录好多份菜单后，同时交给厨师。那么这种方式，对于老板来说人力成本是最低的；对于客人来说，虽然不再享受 VIP 服务并且要等待一定的时间，但是这些都是可接受的；对于服务员来说，基本上他的时间都没有浪费，被老板榨干了每一滴血汗。

如果你是老板，会采用哪种方式呢？

- 到店情况：并发量。到店情况不理想时，一名服务员一份菜单，当然是足够了。所以不同的老板在不同的场合下，将会灵活选择服务员和菜单的配置。
- 客人：客户端请求。
- 点餐内容：客户端发送的实际数据。
- 老板：操作系统。
- 人力成本：系统资源。
- 菜单：文件状态描述符。操作系统对于一个进程能够同时持有的文件状态描述符的个数是有限制的，在 Linux 系统中可用 `$ulimit -n` 命令查看这个限制值，当然也可以（并且应该）进行内核参数调整。
- 服务员：操作系统内核用于网络 I/O 操作的线程（内核线程）。
- 厨师：应用程序线程（厨房就是应用程序进程）。
- 餐单传递方式：包括了阻塞式和非阻塞式两种。
- 方法 A：阻塞式/非阻塞式，同步 I/O。
- 方法 B：使用线程（池）进行处理的阻塞式/非阻塞式同步 I/O。
- 方法 C：多路复用网络 I/O 模型。

#### 7.4.1 典型的多路复用 I/O 实现

多路复用 I/O 模型在应用层工作效率比我们俗称的 BIO 模型快的本质原因是，前者不再使用操作系统级别的“同步 I/O”模型。在 Linux 操作系统环境下，多路复用 I/O 模型就是技术人员通常简称的 NIO 技术。多路复用 I/O 目前具体的实现主要包括四种：select、poll、epoll、



kqueue。表 7-1 是它们的一些重要特性的比较。

表 7-1

I/O 模型	相对性能	关键思路	操作系统	Java 支持情况
select	较高	Reactor	Windows Linux	支持 Reactor 模式（反应器设计模式）。Linux 操作系统的 kernels 2.4 内核版本之前，默认使用 select；而目前 Windows 下对同步 I/O 的支持，使用的都是 select 模型
poll	较高	Reactor	Linux	Linux kernels 2.6 内核版本之前使用 poll 进行支持
epoll	高	Reactor Proactor	Linux	Linux kernels 2.6 内核版本及以后版本使用 epoll 进行支持；Linux kernels 2.6 内核版本之前使用 poll 进行支持；另外一定注意，由于 Linux 下没有 Windows 下的 IOCP 技术提供真正的异步 I/O 支持，所以 Linux 下使用 epoll 模拟异步 I/O
kqueue	高	Proactor	Linux	目前 Java 的版本不支持

多路复用 I/O 技术最适用的是“高并发”场景，所谓高并发是指 1 毫秒内至少同时有成百上千个连接请求准备就绪，其他情况下多路复用 I/O 技术发挥不出它的明显优势。另外使用 Java NIO 进行功能实现，相对于传统的 Socket 套接字实现要复杂一些，所以实际应用中，需要根据自己的业务需求进行技术选择。

### 7.4.2 Java 对多路复用 I/O 技术的支持

多路复用 I/O 技术如图 7-15 所示。

Java 从很早的版本开始提供了对操作系统多路复用技术的调用支持，注意这里提到的是 Java 原生 API 对多路复用的调用支持，而不是各位读者使用过的 Netty 组件。事实上后文会讲到 Netty 或其他网络通信框架是在 Java 原生 API 支持的基础上进一步进行了封装。

接下来我们先给出一个 Java 原生的多路复用框架（Java NIO 框架）实现的服务器端代码，实际上客户端是否使用多路复用 I/O 技术对整个服务端系统架构的性能提升相关性不大。如果你暂时不明白以下代码的全部意义，没关系，本书后文会依次对关键点进行介绍。

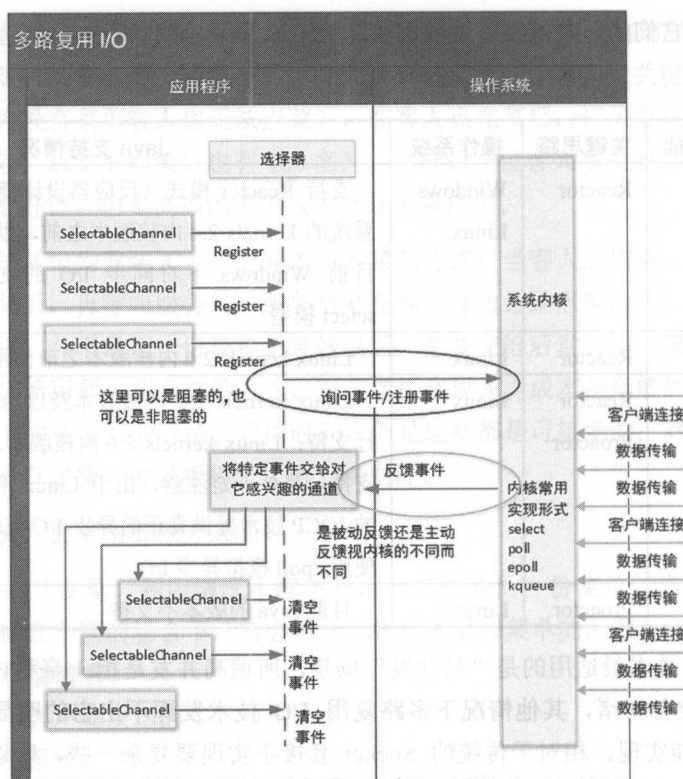


图 7-15 多路复用 I/O

```

.....
import java.nio.ByteBuffer;
import java.nio.channels.SelectableChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
.....
public class SocketServer1 {
    private static final Log LOGGER = LoggerFactory.getLog(SocketServer1.class);

    public static void main(String[] args) throws Exception {
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        // 一定要这样设置
        serverChannel.configureBlocking(false);
        ServerSocket serverSocket = serverChannel.socket();
        serverSocket.setReuseAddress(true);
    }
}

```

```

serverSocket.bind(new InetSocketAddress(83));
Selector selector = Selector.open();
//注意：服务器通道能且只能注册 SelectionKey.OP_ACCEPT 事件
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
try {
    while(true) {
        //如果条件成立，则说明本次询问 selector 并没有获取任何准备好的、感兴趣的事件
        if(selector.select(100) == 0) {
            //=====
            // 这里视业务情况，可以做一些其他业务动作，但实际意义不大
            //=====
            continue;
        }
        //这就是本次询问操作系统所获取的“所关心的事件”的事件类型（每一个通道是独立的）
        Iterator<SelectionKey> selecionKeys = selector.selected
Keys().iterator();
        while(selecionKeys.hasNext()) {
            SelectionKey readyKey = selecionKeys.next();
            // 一定要移除这个已经处理的 readyKey
            // 如果不移除，就会一直存在在 selector.selectedKeys 集合中
            // 待到下一次 selector.select() > 0 时，这个 readyKey 又会被处理一次
            selecionKeys.remove();
            SelectableChannel selectableChannel = readyKey.
Channel();

            if(readyKey.isValid() && readyKey.isAcceptable()) {
                LOGGER.info("==channel 通道已经准备好==");
                // 当 server socket channel 通道已经准备好，
                // 就可以从 server socket channel 中获取 socketchannel 了
                // 拿到 socket channel 后，
                // 要做的事情就是马上到 selector 注册这个 socket channel 感兴趣的事情。
                // 否则无法监听到这个 socket channel 到达的数据
                ServerSocketChannel serverSocketChannel =
(ServerSocketChannel)selectableChannel;
                SocketChannel socketChannel = serverSocketChannel.
accept();

                registerSocketChannel(socketChannel , selector);
            } else if(readyKey.isValid() && readyKey.isReadable
()) {
                SocketServer1.LOGGER.info("==socket channel 数据
准备完成，可以读取==");
                /**
                 * 在 ServerSocketChannel 接收到/准备好一个新的 TCP 连接后
                 * 就会向程序返回一个新的 SocketChannel
                 * 但是这个新的 SocketChannel 并没有在 selector “选择器/代理器”中注册

```

```

    * 所以程序还没法通过 Selector 通知这个 SocketChannel 的事件
    * 于是拿到新的 SocketChannel 后, 要做的第一个事情就是到 selector “选择器/代理
    * 器” 中注册这个 SocketChannel 感兴趣的事件
    * @param selector selector “选择器/代理器”
    */
    private static void registerSocketChannel(SocketChannel socketChannel,
Selector selector) throws Exception {
        socketChannel.configureBlocking(false);
        //socket 通道可以且只可以注册三种事件! SelectionKey.OP_READ |
        //SelectionKey.OP_WRITE | SelectionKey.OP_CONNECT
        socketChannel.register(selector, SelectionKey.OP_READ, ByteBuffer.
allocate(2048));
    }
    //这个方法用于读取从客户端传来的信息
    //并且观察从客户端过来的 socket 通道在经过多次传输后完成传输
    //如果传输完成, 则返回一个 true 的标记
    private static void readSocketChannel(SelectionKey readyKey) throws
Exception {
        SocketChannel clientSocketChannel = (SocketChannel)readyKey.
channel();
        //获取客户端使用的端口
        InetSocketAddress sourceSocketAddress = (InetSocketAddress)
clientSocketChannel.getRemoteAddress();
        Integer resourcePort = sourceSocketAddress.getPort();
        //拿到这个 socket 通道使用的缓存区, 准备读取数据
        //在后文中将详细讲解缓存区的用法概念, 实际上重要的就是三个元素: capacity、
        //position 和 limit
        ByteBuffer contextBytes = (ByteBuffer)readyKey.attachment();
        //将通道的数据写入缓存区
        //由于之前设置了 ByteBuffer 的大小为 2048 byte, 所以可能存在写不完的情况
        //没关系, 本书后文会调整代码, 这里暂时理解为一次接收可以完成
        int realLen = -1;
        try {
            realLen = clientSocketChannel.read(contextBytes);
        } catch (Exception e) {
            //这里抛出了异常, 一般就是客户端因为某种原因终止了。所以关闭 Channel 就行了
            SocketServer1.LOGGER.error(e.getMessage());
            clientSocketChannel.close();
            return;
        }
        //如果缓存区中没有任何数据 (但实际上这个不太可能, 否则就不会触发 OP_READ 事件了)
        if(realLen == -1) {
            SocketServer1.LOGGER.warn("====缓存区没有数据? ====");
            return;
        }
    }

```



```

    }
    //将缓存区从写状态切换为读状态（实际上这个方法是读/写模式互切换）
    //这时 Java NIO 框架中的这个 SocketChannel 的写请求将全部等待
    contextBytes.flip();
    //注意中文乱码的问题，笔者的个人喜好是使用 URLDecoder/URLEncoder 进行解编码
    //当然 Java NIO 框架本身也提供编解码方式，主要看个人使用习惯和所在团队要求
    byte[] messageBytes = contextBytes.array();
    String messageEncode = new String(messageBytes, "UTF-8");
    String message = URLDecoder.decode(messageEncode, "UTF-8");
    //收到了“over”关键字，才会清空 buffer，并回发数据
    //否则不清空缓存，还要还原 buffer 的“写状态”
    if(message.indexOf("over") != -1) {
        //清空已经读取的缓存，并重新切换为写状态(这里要注意 clear() 和
        //capacity() 两个方法的区别)
        contextBytes.clear();
        SocketServer1.LOGGER.info("端口:" + resoucePort + "客户端发来的
信息=====message : " + message);
        //=====
        //接收完成后，就可以在这里正式处理业务了
        //=====
        //回发数据，并关闭 Channel
        ByteBuffer sendBuffer = ByteBuffer.wrap(URLEncoder.encode("回
发处理结果", "UTF-8").getBytes());
        clientSocketChannel.write(sendBuffer);
        clientSocketChannel.close();
    } else {
        SocketServer1.LOGGER.info("端口:" + resoucePort + "客户端信息还
未接收完，继续接收=====message : " + message);
        //这时，limit 和 capacity 的值一致，position 的位置是 realLen 的位置
        contextBytes.position(realLen);
        contextBytes.limit(contextBytes.capacity());
    }
}
}

```

以上代码片段中注释是比较清楚的，但还是要对几个关键点进行一下讲解。

- `serverChannel.register(Selector sel, int ops, Object att)`: 实际上 `register(Selector sel, int ops, Object att)` 方法是 `ServerSocketChannel` 类的父类 `AbstractSelectableChannel` 提供的一个方法，也就是说只要继承了 `AbstractSelectableChannel` 类的子类就可以注册到选择器中。通过观察整个 `AbstractSelectableChannel` 继承关系，图 7-16 中的这些类可以被注册到选择器中。

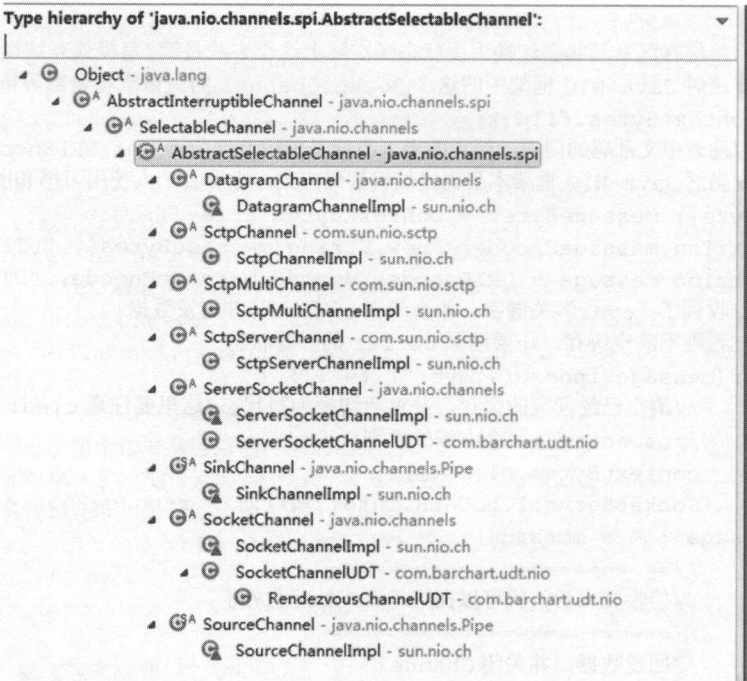


图 7-16 可被注册到选择器的类

- `SelectionKey.OP_ACCEPT`: 不同的 `Channel` 对象可以注册的“关心的事件”是不一样的。例如 `ServerSocketChannel` 除了能够被允许关注 `OP_ACCEPT` 事件，不允许再关心其他事件。如表 7-2 所示梳理了常使用的 `AbstractSelectableChannel` 子类可以注册的事件列表。

表 7-2

通道类	通道作用	可关注的事件
<code>ServerSocketChannel</code>	服务器端通道	<code>SelectionKey.OP_ACCEPT</code>
<code>DatagramChannel</code>	UDP 协议通道	<code>SelectionKey.OP_READ</code> <code>SelectionKey.OP_WRITE</code>
<code>SocketChannel</code>	TCP 协议通道	<code>SelectionKey.OP_READ</code> <code>SelectionKey.OP_WRITE</code> <code>SelectionKey.OP_CONNECT</code>

实际上，通过每一个 `AbstractSelectableChannel` 子类所实现的 `public final int validOps()` 方法，可以查看这个通道“可以关心的 I/O 事件”。

- `selector.selectedKeys().iterator()`: 当选择器 `Selector` 收到操作系统的 I/O 操作事件后，它



的 `selectedKeys` 将在下一次轮询操作中，收到这些事件的关键描述字（不同的 `Channel`，就算关键字一样，也会存储成两个对象）。但是每一个“事件关键字”被处理后都必须移除，否则下一次轮询时，这个事件会被重复处理。

Returns this selector's selected-key set.

Keys may be removed from, but not directly added to, the selected-key set. Any attempt to add an object to the key set will cause an `UnsupportedOperationException` to be thrown.

The selected-key set is not thread-safe.

### 1. 重要概念：Channel

`Channel` 通道，是一个用来完成应用程序和操作系统交互事件、传递内容的渠道，注意是连接到操作系统。一个通道会有一个专属的文件状态描述符。既然是和操作系统进行内容的传递，那就说明应用程序可以通过通道从操作系统读取数据，也可以通过通道向操作系统写数据。JDK API 中的 `Channel` 的描述是：

A channel represents an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct io operations, for example reading or writing.

A channel is either open or closed. A channel is open upon creation, and once closed it remains closed. Once a channel is closed, any attempt to invoke an io operation upon it will cause a `ClosedChannelException` to be thrown. Whether or not a channel is open may be tested by invoking its `isOpen` method.

在 Java NIO 框架中，原生的 `Channel` 通道实现如图 7-17 所示。

所有被 `Selector`（选择器）注册的通道，只能是继承了 `SelectableChannel` 类的子类。其中有几个关键的 `Channel` 通道实现，需要在这里进行说明。

- `ServerSocketChannel`：应用服务器端程序的监听通道。只有通过这个通道，应用程序才能向操作系统注册支持“多路复用 I/O”的端口监听。它同时支持 UDP 协议和 TCP 协议。
- `SocketChannel`：TCP Socket 套接字的监听通道，一个 `Socket` 套接字对应了一个客户端（IP 和端口）到服务器端（IP 和端口）的通信连接。
- `DatagramChannel`：UDP 数据报文的监听通道。

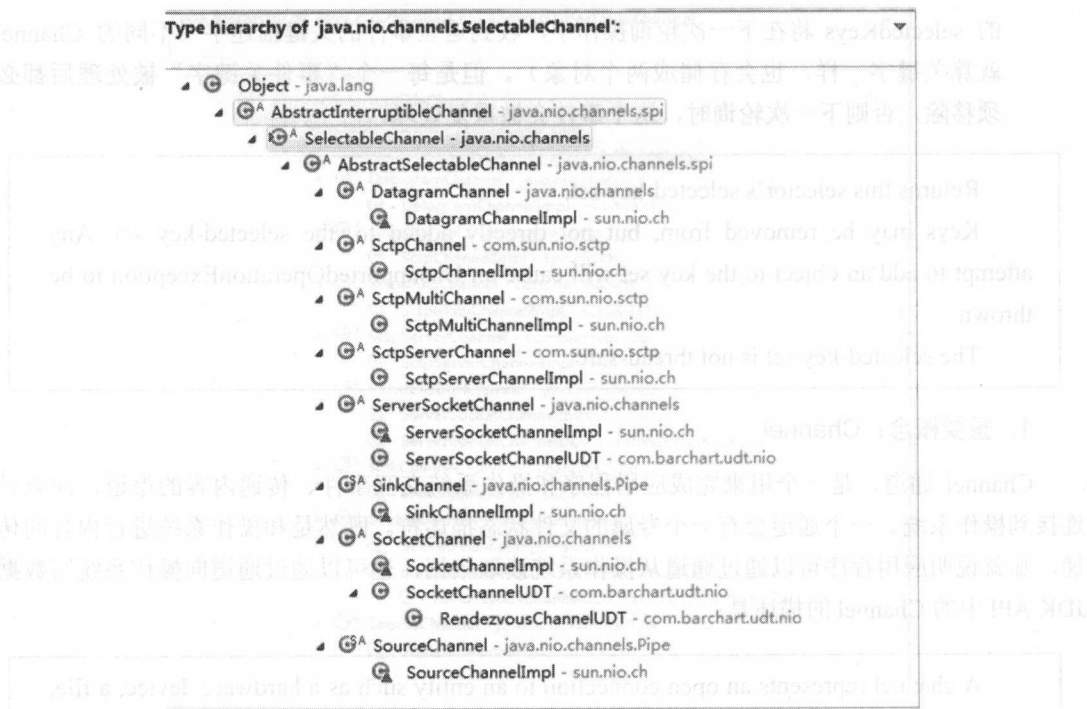


图 7-17 Channel 通道实现

## 2. 重要概念: Buffer

**数据缓存区:** 在 Java 原生 NIO 框架中, 为了保证每个通道的数据读/写速度, Java NIO 框架为每一种需要支持数据读/写的通道集成了 Buffer 的支持。

这句话怎么理解呢? 例如 ServerSocketChannel 通道只支持对 OP\_ACCEPT 事件的监听, 它是不能直接进行网络数据内容的读/写的, 所以 ServerSocketChannel 是没有集成 Buffer 的。

Buffer 有两种工作模式: 写模式和读模式。在读模式下, 应用程序只能从 Buffer 中读取数据, 不能进行写操作。但是在写模式下, 应用程序是可以进行读操作的, 这就表示可能会出现脏读的情况。所以一旦决定要从 Buffer 中读取数据, 就一定要将 Buffer 的状态改为读模式。如图 7-18 所示。

- position: 缓存区目前正在操作的数据块位置。
- limit: 缓存区最大可以进行操作的位置。缓存区的读/写状态正是由这个属性控制的。
- capacity: 缓存区的最大容量。这个容量是在缓存区创建时进行指定的。由于高并发时通道数量往往会很庞大, 所以每一个缓存区的容量最好不要过大。在下文 Java NIO 框架的代码实例中, 我们将进行 Buffer 缓存区操作的演示。

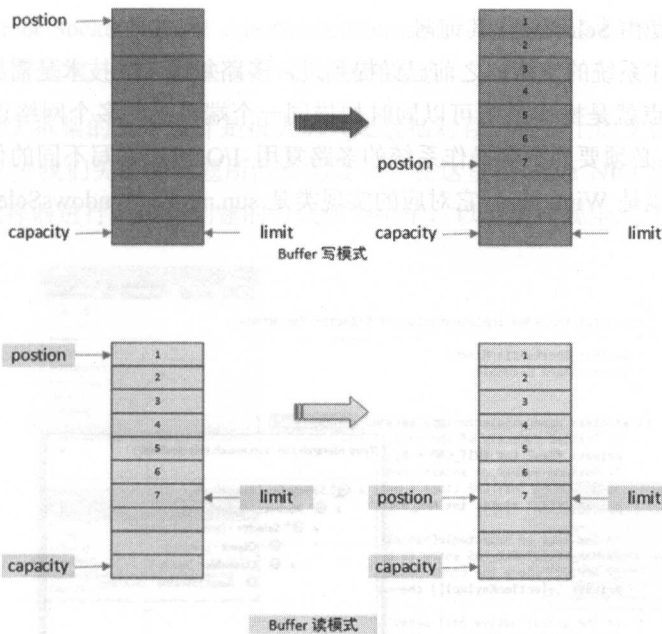


图 7-18 Buffer 的写模式和读模式

### 3. 重要概念：Selector

Selector 的英文含义是“选择器”，不过根据我们详细介绍的 Selector 的岗位职责，你可以把它称为“轮询代理器”、“事件订阅器”、“Channel 容器管理机”，等等。

- 事件订阅和 Channel 管理：应用程序将向 Selector 对象注册需要它关注的 Channel，以及具体的某一个 Channel 会对哪些 I/O 事件感兴趣。Selector 中也会维护一个“已经注册的 Channel”的容器。以下代码来自 WindowsSelectorImpl 实现类中，对已经注册的 Channel 的管理容器：

```
// Initial capacity of the poll array
private final int INIT_CAP = 8;
// Maximum number of sockets for select().
// Should be INIT_CAP times a power of 2
private final static int MAX_SELECTABLE_FDS = 1024;

// The list of SelectableChannels serviced by this Selector. Every mod
// MAX_SELECTABLE_FDS entry is bogus, to align this array with the poll
// array, where the corresponding entry is occupied by the wakeupSocket
private SelectionKeyImpl[] channelArray = new SelectionKeyImpl[INIT_CAP];
```

- 轮询代理：应用层不再通过阻塞模式或者非阻塞模式直接询问操作系统“事件有没有

发生”，而是由 Selector 代其询问。

- 实现不同操作系统的支持：之前已经提到过，多路复用 I/O 技术是需要操作系统进行支持的，其特点就是操作系统可以同时扫描同一个端口上的多个网络连接。所以作为上层的 JVM，必须要为不同操作系统的多路复用 I/O 实现编写不同的代码。同样笔者使用的测试环境是 Windows，它对应的实现类是 sun.nio.ch.WindowsSelectorImpl，如图 7-19 所示。

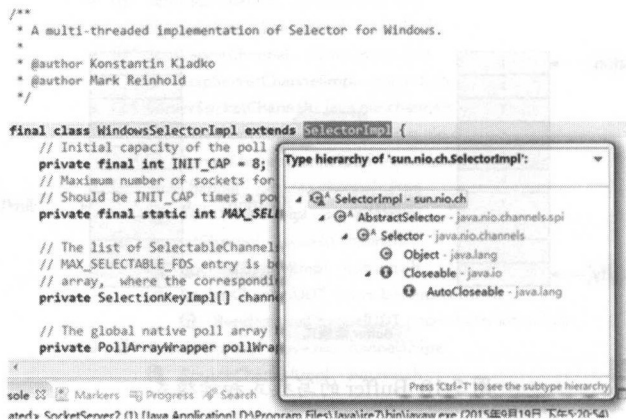


图 7-19 WindowsSelectorImpl 源码截图

### 7.4.3 Java NIO 框架简要设计分析

通过前面的描述，我们知道了多路复用 I/O 技术是操作系统的内核实现。在不同的操作系统甚至同一系列操作系统的版本中，所实现的多路复用 I/O 技术都是不一样的。那么作为跨平台的 Java JVM 来说，如何适应多种多样的多路复用 I/O 技术实现呢？这时面向对象的威力就显现出来了：无论使用哪种实现方式，它们都会有“选择器”、“通道”、“缓存”这几个操作要素，那么可以为不同的多路复用 I/O 技术创建一个统一的抽象组，并且为不同的操作系统进行具体的实现。Java NIO 中对各种多路复用 I/O 的支持，主要的基础是 java.nio.channels.spi.SelectorProvider 抽象类，其中的几个主要抽象方法如下。

- public abstract DatagramChannel openDatagramChannel(): 创建和这个操作系统匹配的 UDP 通道实现。
- public abstract AbstractSelector openSelector(): 创建和这个操作系统匹配的 NIO 选择器。就像上文所述，不同的操作系统、不同的版本所默认支持的 NIO 模型是不一样的。
- public abstract ServerSocketChannel openServerSocketChannel(): 创建和这个 NIO 模型匹配的服务器端通道。

- `public abstract SocketChannel openSocketChannel()`: 创建和这个 NIO 模型匹配的 TCP Socket 套接字通道，用来反映客户端的 TCP 连接。

由于 Java NIO 框架的整个设计是很大的（还包括对存储设备上的文件进行操作），所以本书只能还原一部分我们关心的问题所涉及的设计。在这里以 Java NIO 框架中对于不同多路复用 I/O 技术的选择器进行实例化创建的方式作为例子，以便窥斑观全局（图 7-20）。

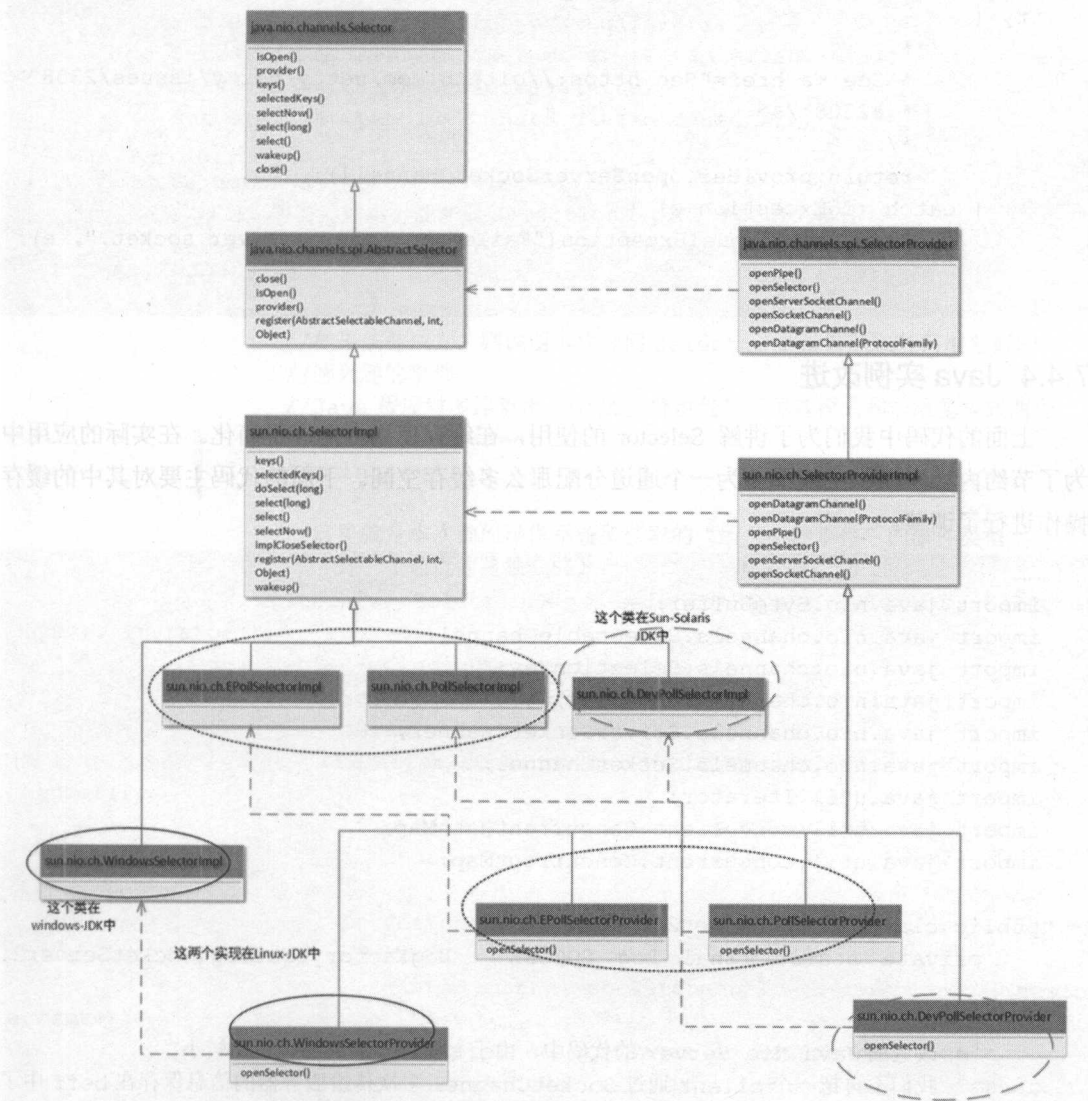


图 7-20 Java NIO 部分设计结构



很明显，不同的 `SelectorProvider` 实现对应了不同的选择器，由具体的 `SelectorProvider` 实现进行创建。另外说明一下，实际上 `Netty` 底层也通过这个设计获得具体使用的 `NIO` 模型，后面讲解 `Netty` 时会讲到这个问题。以下代码是 `Netty 4.0` 中 `NIOServerSocketChannel` 进行实例化时的核心代码片段：

```
private static ServerSocketChannel newSocket(SelectorProvider provider)
{
    try {
        /**
         * See <a href="See https://github.com/netty/netty/issues/2308">
         * #2308</a>.
         */
        return provider.openServerSocketChannel();
    } catch (IOException e) {
        throw new ChannelException("Failed to open a server socket.", e);
    }
}
```

#### 7.4.4 Java 实例改进

上面的代码中我们为了讲解 `Selector` 的使用，在缓存使用上进行了简化。在实际的应用中，为了节约内存资源，一般不会为一个通道分配那么多缓存空间。下面的代码主要对其中的缓存操作进行了调整：

```
.....
import java.nio.ByteBuffer;
import java.nio.channels.SelectableChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
.....

public class SocketServer2 {
    private static final Log LOGGER = LoggerFactory.getLog(SocketServer2.
class);
    /**
     * 改进的 Java Nio Server 的代码中，由于 Buffer 的大小设置比较小
     * 我们不再把一个 client 通过 SocketChannel 多次传给服务器的信息保存在 beff 中了
     * （因为根本存不下）
     * 我们使用 SocketChanel 的 hashCode 作为 key（当然你也可以自己确定一个 id），
```



```

* 使用信息的 stringBuffer 作为 value, 将信息存储到服务器端的一个内存区域 HC 中
*
* 如果你不清楚 ConcurrentHashMap 的作用和工作原理, 请查询相关资料
*/
private static final Map<Integer, StringBuffer> MESSAGEHASHCONTEXT =
new ConcurrentHashMap<>();
public static void main(String[] args) throws Exception {
    ServerSocketChannel serverChannel = ServerSocketChannel.open();
    serverChannel.configureBlocking(false);
    ServerSocket serverSocket = serverChannel.socket();
    serverSocket.setReuseAddress(true);
    serverSocket.bind(new InetSocketAddress(83));

    Selector selector = Selector.open();
    //注意: 服务器通道只能注册 SelectionKey.OP_ACCEPT 事件
    serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    try {
        while(true) {
            //如果条件成立, 则说明本次询问 Selector 并没有获取任何准备好的、
            //感兴趣的事件
            //Java 程序对多路复用 I/O 的支持也包括了阻塞模式和非阻塞模式两种
            if(selector.select(100) == 0) {
                continue;
            }
            //这里就是本次询问操作系统所获取的“所关心的事件”的事件类型
            //(每一个通道都是独立的)
            Iterator<SelectionKey> selecionKeys = selector.Selected
Keys(). iterator();
            while(selecionKeys.hasNext()) {
                SelectionKey readyKey = selecionKeys.next();
                selecionKeys.remove();
                SelectableChannel selectableChannel = readyKey.
channel();

                if(readyKey.isValid() && readyKey.isAcceptable()) {
                    SocketServer2.LOGGER.info("=====channel 通道已经
准备好=====");
                    ServerSocketChannel serverSocketChannel = (Server
SocketChannel)selectableChannel;
                    SocketChannel socketChannel = serverSocketChannel.
accept();

                    registerSocketChannel(socketChannel , selector);
                }else if(readyKey.isValid() && readyKey.isConnec
table()) {

                    SocketServer2.LOGGER.info("=====socket channel

```

```

建立连接=====");
        } else if (readyKey.isValid() && readyKey.isReadable()) {
            SocketServer2.LOGGER.info("=====socket channel
数据准备完成，可以去读==读取=====");
            readSocketChannel(readyKey);
        }
    }
}
} catch (Exception e) {
    SocketServer2.LOGGER.error(e.getMessage(), e);
} finally {
    serverSocket.close();
}
}

private static void registerSocketChannel(SocketChannel socketChannel,
Selector selector) throws Exception {
    socketChannel.configureBlocking(false);
    //Socket 通道可以且只可以注册三种事件: SelectionKey.OP_READ |
    //SelectionKey.OP_WRITE | SelectionKey.OP_CONNECT
    //最后一个参数视为这个 socketchannel 分配的缓存区
    socketChannel.register(selector, SelectionKey.OP_READ, ByteBuffer.
allocate(50));
}
/**
 * 这个方法用于读取从客户端传来的信息
 * 并且观察从客户端过来的 SocketChannel 在经过多次传输后完成传输
 * 如果传输完成，则返回一个 true 的标记
 */
private static void readSocketChannel(SelectionKey readyKey) throws
Exception {
    SocketChannel clientSocketChannel = (SocketChannel)readyKey.
channel();
    //获取客户端使用的端口
    InetSocketAddress sourceSocketAddress = (InetSocketAddress)
clientSocketChannel.getRemoteAddress();
    Integer resourcePort = sourceSocketAddress.getPort();
    //拿到这个 Socket Channel 使用的缓存区，准备读取数据
    //后文将详细讲解缓存区的用法概念，实际上重要的就是三个元素: capacity、
    //position 和 limit
    ByteBuffer contextBytes = (ByteBuffer)readyKey.attachment();
    //将通道的数据写入缓存区
    //这次为了演示 Buff 的使用方式，我们故意将 Buff 的容量缩小到 50 byte，
    //以便演示 Channel 对 Buff 的多次读/写操作
    int realLen = 0;

```

```

StringBuffer message = new StringBuffer();
//这句话的意思是，将目前通道中的数据写入缓存区
//最大可写入的数据量就是 Buff 的容量
while((realLen = clientSocketChannel.read(contextBytes)) != 0) {
    //一定要把 Buffer 切换成“读”模式，否则由于 limit = capacity
    //在 read 没有写满的情况下，就会导致多读
    contextBytes.flip();
    int position = contextBytes.position();
    int capacity = contextBytes.capacity();
    byte[] messageBytes = new byte[capacity];
    contextBytes.get(messageBytes, position, realLen);

    //这种方式也是可以读取数据的，而且不用关心 position 的位置
    //因为目前 contextBytes 所有的数据全部转出为一个 byte 数组
    //使用这种方式时，一定要自己控制好读取的最终位置（realLen 很重要）
    //byte[] messageBytes = contextBytes.array();
    //注意中文乱码的问题，笔者个人喜好使用 URLDecoder/URLEncoder 进行解
    //编码
    //当然 Java NIO 框架本身也提供编解码方式，看个人使用习惯
    String messageEncode = new String(messageBytes, 0, realLen,
"UTF-8");

    message.append(messageEncode);
    //再切换成“写”模式，直接存入缓存的方式最快捷
    contextBytes.clear();
}
//如果发现本次接收的信息中有“over”关键字，说明信息接收完成
if(URLDecoder.decode(message.toString(),"UTF-8").indexOf ("over")!=
-1) {
    //则从 messageHashContext 中，取出之前已经收到的信息，组合成完整的信息
    Integer channelUUID = clientSocketChannel.hashCode();
    SocketServer2.LOGGER.info("端口:" + resoucePort + "客户端发来的
信息=====message : " + message);
    StringBuffer completeMessage;
    //清空 MESSAGEHASHCONTEXT 中的历史记录
    StringBuffer historyMessage = MESSAGEHASHCONTEXT. Remove
(channelUUID);
    if(historyMessage == null) {
        completeMessage = message;
    } else {
        completeMessage = historyMessage.append(message);
    }
    SocketServer2.LOGGER.info("端口:" + resoucePort + "客户端发来的
完整信息=====completeMessage : " + URLDecoder.decode(completeMessage.
toString(), "UTF-8"));
}

```



```

//=====
// 接收完成后, 可以在这里正式处理业务了
//=====
//回发数据, 并关闭 Channel
ByteBuffer sendBuffer = ByteBuffer.wrap(URLEncoder.encode("回
发处理结果", "UTF-8").getBytes());
clientSocketChannel.write(sendBuffer);
clientSocketChannel.close();
} else {
    //如果没有发现“over”关键字, 说明还没有接收完, 则将本次接收到的信息存
    //入 MESSAGEHASHCONTEXT
    SocketServer2.LOGGER.info("端口:" + resoucePort + "客户端信息还
未接收完, 继续接收====message : " + URLDecoder.decode(message.toString(),
"UTF-8"));
    //每一个 Channel 对象都是独立的, 所以可以使用对象的 Hash 值, 作为唯一标识
    Integer channelUUID = clientSocketChannel.hashCode();
    //然后获取这个 Channel 下以前已经到达的信息
    StringBuffer historyMessage = MESSAGEHASHCONTEXT. Get
(channelUUID);
    if(historyMessage == null) {
        historyMessage = new StringBuffer();
        MESSAGEHASHCONTEXT.put(channelUUID,
            historyMessage.append(message));
    }
    historyMessage.append(message);
}
}
}

```

以上代码应该没有过多需要讲解的了。当然, 你还是可以加入线程池技术进行具体的业务处理。注意, 如果读者要将以上代码思维应用到工作环境, 那么就一定要使用线程池, 因为这样可以保证线程规模的可控性。当然笔者更建议使用 Netty 这样的组件, 原因会在后文进行讲述。

## 7.4.5 多路复用 I/O 的优缺点

多路复用 I/O 技术由操作系统提供支持, 并提供给各种高级语言进行使用。它针对阻塞式同步 I/O 和非阻塞式同步 I/O 而言有很多优势, 最直接的效果就是它绕过了 I/O 在操作系统层面的 accept() 方法的阻塞问题。

- 使用多路复用 I/O 技术后, 应用程序就可以不用再单纯使用多线程技术来解决并发 I/O 处理的性能问题了 (针对操作系统内核 I/O 管理模块和应用程序进程而言都是这样的)。

在实际业务的处理中，应用程序进程还是需要引入（由线程池支持的）多线程技术的。

- 同一个端口可以处理多种网络协议。例如，使用 `ServerSocketChannel` 类的服务器端口监听，既可以接收到 TCP 协议又可以接收到 UDP 协议内容。也就是说端口的数据接收规则只和 `Selector` 注册的需要关心的事件有关。
- 操作系统级别的优化：多路复用 I/O 技术可以使操作系统级别在一个端口上能够同时接受多个客户端的 I/O 事件，同时具有之前我们讲到的阻塞式同步 I/O 和非阻塞式同步 I/O 的所有特点。`Selector` 的一部分作用更相当于“轮询代理器”。
- 都是同步 I/O 模型：目前我们介绍的阻塞式 I/O、非阻塞式 I/O，甚至包括多路复用 I/O，这些都是基于操作系统级别对“同步 I/O”的实现。我们一直在说“同步 I/O”，一直都没有详细说什么叫作“同步 I/O”。实际上一句话就可以解释清楚：只有上层（包括上层的某种代理机制）系统询问“我”是否有某个事件发生了，否则“我”不会主动告诉上层系统事件发生了。

## 7.5 网络 I/O 模型：异步 I/O

以上章节中，我们分别讨论了阻塞式同步 I/O、非阻塞式同步 I/O、多路复用 I/O 这三种 I/O 通信模型，以及 Java 语言对于这三种 I/O 模型的支持。重点说明了 I/O 模型是由操作系统提供支持的，且这三种 I/O 模型都是同步 I/O 的范畴，都是采用“应用程序不询问‘我’，‘我’绝不会主动通知”的方式。

异步 I/O（又称为 AIO）则是采用“订阅—通知”工作模式：即应用程序向操作系统注册 I/O 监听，然后继续做自己的事情。当操作系统发生 I/O 事件，并且准备好数据后，再主动通知应用程序，触发相应的函数，如图 7-21 所示。

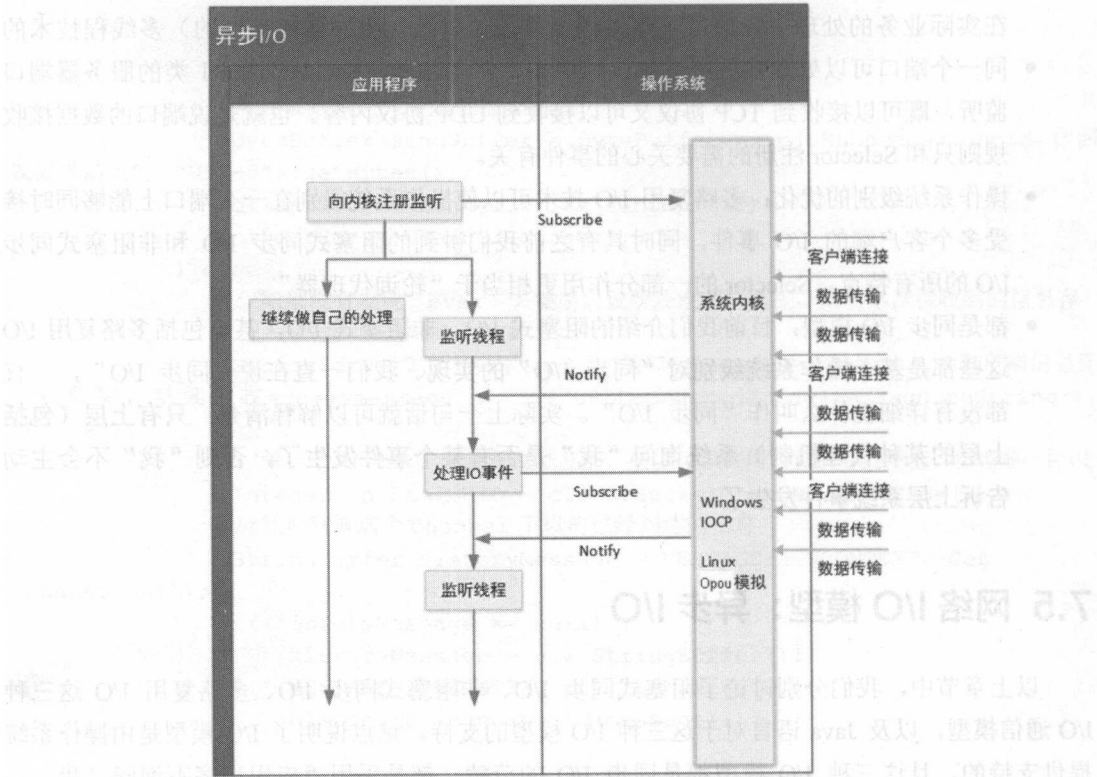


图 7-21 异步 I/O 工作模型

和上文介绍的三种同步 I/O 模型类似，异步 I/O 也必须由操作系统进行支持。微软的 Windows 系统提供了一种异步 I/O 技术：IOCP（I/O Completion Port，I/O 完成端口）；多个 Linux 操作系统版本（例如 CentOS 5x 等）下由于没有这种异步 I/O 技术，所以使用的是 epoll（上文介绍过的一种多路复用 I/O 技术的实现）对异步 I/O 进行模拟。

7.5.1 Java 对 AIO 的支持

图 7-22 中给出了 Java AIO 框架的部分设计结构，前面已经提到 Java AIO 框架在 Windows 下使用 Windows IOCP 技术，在多个版本的 Linux 操作系统下（例如 CentOS 5x 版本）使用 epoll 多路复用 I/O 技术模拟异步 I/O。这从 Java AIO 框架的部分类的设计上就可以看出来，例如框架中，在 Windows 下负责实现套接字通道的具体类是“sun.nio.ch.WindowsAsynchronousSocketChannelImpl”，其引用的 IOCP 类型文档注释如下：



Windows implementation of AsynchronousChannelGroup encapsulating an I/O completion port.

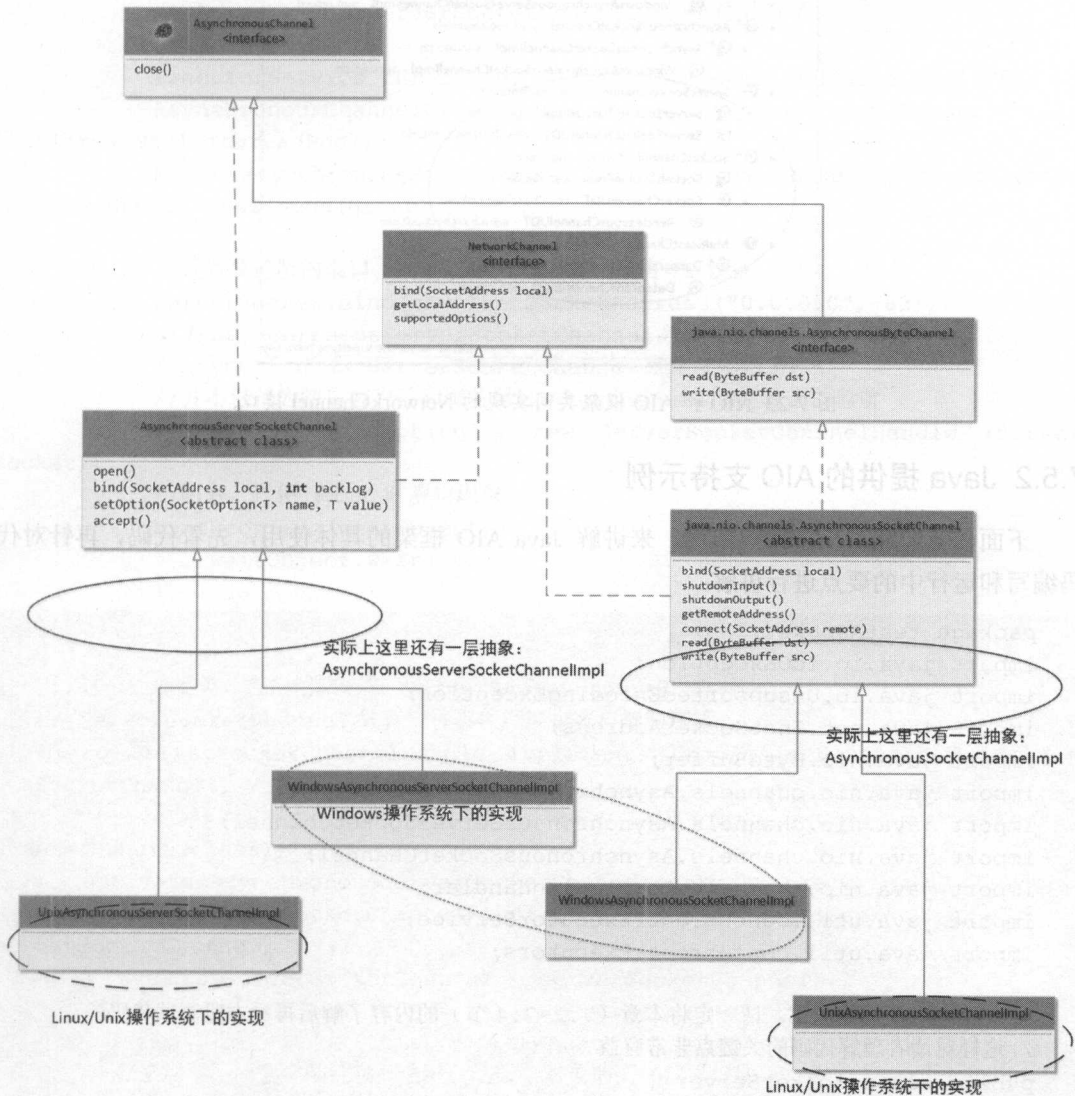


图 7-22 Java-AIO 框架

特别说明一下，请注意图 7-22 中的“java.nio.channels.NetworkChannel”接口，这个接口同样被 Java NIO 框架实现了，如图 7-23 所示。

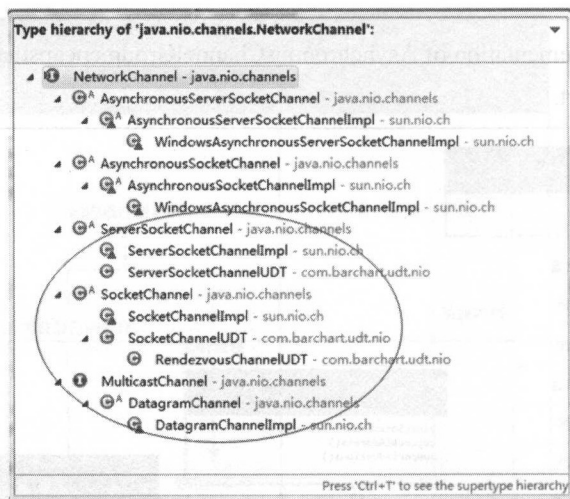


图 7-23 NIO 和 AIO 框架共同实现的 NetworkChannel 接口

## 7.5.2 Java 提供的 AIO 支持示例

下面，我们通过一个代码示例，来讲解 Java AIO 框架的具体使用，先看代码，再针对代码编写和运行中的要点进行讲解：

```
package testASocket;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousChannelGroup;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
.....
//Java AIO 框架测试。请一定将本章（7.2~7.4 节）的内容了解后再看本段测试代码
//这样对读者理解代码的关键点非常有益
public class SocketServer {
    private static final Object waitObject = new Object();
    public static void main(String[] args) throws Exception {
        /*
        * 对于使用的线程池技术，笔者需要多说几句
        * 1. Executors 是线程池生成工具，通过这个工具我们可以很轻松地生成“固定大
        * 小的线程池”、“调度池”、“可伸缩线程数量的池”。具体请看 API 文档
```

```

    * 2. 当然你也可以通过 ThreadPoolExecutor 直接实例化线程池
    * 3. 这个线程池是用来得到操作系统的“I/O 事件通知”的，不是用来进行“得到 I/O
    * 数据后的业务处理的”。要进行后者的操作，你可以再使用另一个池（最好不要混用）
    * 4. 你也可以不使用线程池（不推荐），如果决定不使用线程池，那么直接调用
    * AsynchronousServerSocketChannel.open() 就行了
    * */
    ExecutorService threadPool = Executors.newFixedThreadPool(20);
    AsynchronousChannelGroup group = AsynchronousChannelGroup.
withThreadPool (threadPool);
    Final AsynchronousServerSocketChannel serverSocket = Asynchronous
ServerSocketChannel.open(group);

    //设置要监听的端口“0.0.0.0”代表本机所有 IP 设备
    serverSocket.bind(new InetSocketAddress("0.0.0.0", 83));
    //为 AsynchronousServerSocketChannel 注册监听，注意只是为
    //Asynchronous ServerSocketChannel 通道注册监听
    //并不包括为随后客户端和服务端 socketchannel 通道注册的监听
    serverSocket.accept(null, new ServerSocketChannelHandle (server
Socket));

    //阻塞，以便 debug 时观察现象
    synchronized(waitObject) {
        waitObject.wait();
    }
}

//这个处理器类，专门用来响应 ServerSocketChannel 的事件。
//ServerSocketChannel 只有一种事件：接收客户端的连接
Class ServerSocketChannelHandle implements CompletionHandler< Asynchron
ousSocketChannel, Void> {
    private static final Log LOGGER = LogFactory.getLog (ServerSocket
ChannelHandle.class);
    private AsynchronousServerSocketChannel serverSocketChannel;
    public ServerSocketChannelHandle (AsynchronousServerSocketChannel
serverSocketChannel) {
        this.serverSocketChannel = serverSocketChannel;
    }
    //注意，我们分别查看 this、socketChannel、attachment 三个对象的 ID
    //来观察不同客户端连接到达时，这三个对象的变化，以说明
ServerSocketChannelHandle 的监听模式
    @Override
    public void completed(AsynchronousSocketChannel socketChannel, Void
attachment) {
        ServerSocketChannelHandle.LOGGER.info("completed
(AsynchronousSocketChannel result, ByteBuffer attachment)");

```

```

        this.serverSocketChannel.accept(attachment, this);
        //为这个新的 SocketChannel 注册“read”事件，以便操作系统在收到数据并准备
        //好后，主动通知应用程序
        //在这里，由于我们要将这个客户端多次传输的数据累加起来一起处理，所以将一个
        //stringbuffer 对象作为一个“附件”依附在这个 Channel 上
        ByteBuffer readBuffer = ByteBuffer.allocate(50);
        socketChannel.read(readBuffer, new StringBuffer(), new Socket
ChannelReadHandle(socketChannel, readBuffer));
    }
    @Override
    public void failed(Throwable exc, Void attachment) {
        ServerSocketChannelHandle.LOGGER.info("failed(Throwable exc,
ByteBuffer attachment)");
    }
}
/**
 * 负责对每一个 SocketChannel 的数据获取事件进行监听。
 * 重要说明：每个 SocketChannel 都会有一个独立工作的 SocketChannelReadHandle 对象
 * (CompletionHandler 接口的实现)
 * 其中又都将独享一个“文件状态标识”对象 FileDescriptor
 * 还有一个独立的由程序员定义的 Buffer 缓存（这里我们使用的是 ByteBuffer）
 * 所以不用担心在服务器端会出现“窜对象”这种情况，因为 Java AIO 框架已经帮你组织好了
 *
 * 另一个重要的点是，用于生成 Channel 的对象 AsynchronousChannelProvider 是单例模
 * 式，无论在哪组 socketchannel，都是一个对象引用
 */
Class SocketChannelReadHandle implements CompletionHandler<Integer,
StringBuffer> {
    Private static final Log LOGGER = LogFactory.getLog(Socket Channel
ReadHandle.class);
    private AsynchronousSocketChannel socketChannel;
    /**
     * 专门用于进行这个通道数据缓存操作的 ByteBuffer
     * 当然，你也可以作为 CompletionHandler 的 attachment 形式传入
     */
    private ByteBuffer byteBuffer;
    public SocketChannelReadHandle(AsynchronousSocketChannel socketChannel,
ByteBuffer byteBuffer) {
        this.socketChannel = socketChannel;
        this.byteBuffer = byteBuffer;
    }
    @Override
    public void completed(Integer result, StringBuffer historyContext) {
        //如果条件成立，说明客户端主动终止了 TCP 套接字，这时服务器端终止就可以了

```



```

        if(result == -1) {
            try {
                this.socketChannel.close();
            } catch (IOException e) {
                SocketChannelReadHandle.LOGGER.error(e);
            }
            return;
        }
        SocketChannelReadHandle.LOGGER.info("completed(Integer    result,
Void attachment) : 然后我们来取出通道中准备好的值");
        /*
        * 实际上, 由于我们从 Integer result 知道了本次 Channel 从操作系统获取数据
        * 总长度
        * 所以我们不需要切换成“读模式”, 但是为了保证编码的规范性, 还是建议进行切换
        *
        * 另外, 无论是 Java AIO 框架还是 Java NIO 框架, 都会出现“Buffer 的总容量”
        * 另外小于“当前从操作系统获取到的总数据量”的情况
        * 但区别是, Java AIO 框架不需要专门考虑处理这样的情况, 因为 Java AIO 框架
        * 另外已经帮我们做了处理
        */
        this.byteBuffer.flip();
        byte[] contexts = new byte[1024];
        this.byteBuffer.get(contexts, 0, result);
        this.byteBuffer.clear();
        try {
            String nowContent = new String(contexts , 0 , result , "UTF-
8");
            historyContext.append(nowContent);
            SocketChannelReadHandle.LOGGER.info("===== 目前的传
输结果: " + historyContext);
        } catch (UnsupportedEncodingException e) {
            SocketChannelReadHandle.LOGGER.error(e);
        }
        //如果条件成立, 则说明还没有接收到“结束标记”
        if(historyContext.indexOf("over") == -1) {
            return;
        }
        //和前文出现的代码相同, 我们以“over”符号作为客户端完整信息的标记
        //如何较好地处理“粘包”问题, 下一节马上进行介绍
        SocketChannelReadHandle.LOGGER.info("==: 收到完整信息, 开始处理业务
==");
        historyContext = new StringBuffer();
        //还要继续监听(一次监听一次通知)
        this.socketChannel.read(this.byteBuffer, historyContext, this);

```



```

    }
    @Override
    public void failed(Throwable exc, StringBuffer historyContext) {
        LOGGER.info("====发现客户端异常关闭，服务器将关闭 TCP 通道");
        try {
            this.socketChannel.close();
        } catch (IOException e) {
            SocketChannelReadHandle.LOGGER.error(e);
        }
    }
}

```

注意，在 Java NIO 框架中，我们说到了一个重要概念“Selector”（选择器）。它负责代替应用查询中所有已注册的通道到操作系统中进行 I/O 事件轮询、管理当前注册的通道集合、定位发生事件的通道等操作；但是在 Java AIO 框架中，由于应用程序不是“轮询”方式，而是“订阅—通知”方式，所以不再需要“Selector”（选择器）了，改由 **Channel** 通道直接到操作系统注册监听。

在 Java AIO 框架中，只实现了两种网络 I/O 通道“**AsynchronousServerSocketChannel**”（服务器监听通道）、“**AsynchronousSocketChannel**”（Socket 套接字通道）。但是无论哪种通道它们都有独立的 **fileDescriptor**（文件标识符）、**attachment**（附件，可以是任意对象，类似“通道上下文”），并被独立的 **SocketChannelReadHandle** 类实例引用。我们通过 debug 操作来看看它们的引用结构。

在测试过程中，启动了两个客户端（客户端用什么语言来写都行，用阻塞或者非阻塞方式也都没问题，只要支持 TCP Socket 套接字功能的就行），然后我们观察服务器端对这两个客户端通道的处理情况，如图 7-24 所示。



图 7-24 客户端启动效果

可以看到，在服务器端分别为客户端 1 和客户端 2 创建的两个 **WindowsAsynchronousSocketChannelImpl** 对象（图 7-25），分别为：

客户端 1: WindowsAsynchronousSocketChannelImpl: 760 | FileDescriptor: 762

客户端 2: WindowsAsynchronousSocketChannelImpl: 792 | FileDescriptor: 797

<ul style="list-style-type: none"> <li>socketChannel           <ul style="list-style-type: none"> <li>closeLock</li> <li>completionKey</li> <li>fd               <ul style="list-style-type: none"> <li>handle</li> </ul> </li> <li>ioCache</li> <li>ioep</li> </ul> </li> </ul>	WindowsAsynchronousSocketChannelImpl (id=760) ReentrantReadWriteLock (id=761) 6 FileDescriptor (id=762) 1488 PendingIoCache (id=763) Ioep (id=22)
<ul style="list-style-type: none"> <li>socketChannel           <ul style="list-style-type: none"> <li>closeLock</li> <li>completionKey</li> <li>fd               <ul style="list-style-type: none"> <li>handle</li> </ul> </li> <li>ioCache</li> <li>ioep</li> </ul> </li> </ul>	WindowsAsynchronousSocketChannelImpl (id=792) ReentrantReadWriteLock (id=796) 7 FileDescriptor (id=797) 1592 PendingIoCache (id=798) Ioep (id=22)

图 7-25 客户端的对象情况 (1)

接下来，我们让两个客户端发送信息到服务器端，并观察服务器端的处理情况。客户端 1 发来的消息和客户端 2 发来的消息在服务器端的处理情况如图 7-26 所示。

<ul style="list-style-type: none"> <li>this           <ul style="list-style-type: none"> <li>byteBuffer</li> <li>socketChannel</li> <li>result               <ul style="list-style-type: none"> <li>value</li> </ul> </li> <li>historyContext               <ul style="list-style-type: none"> <li>count</li> <li>value</li> </ul> </li> </ul> </li> </ul>	SocketChannelReadHandle (id=803) HeapByteBuffer (id=808) WindowsAsynchronousSocketChannelImpl (id=760) Integer (id=81) 50 StringBuffer (id=807) 0 (id=812)
<ul style="list-style-type: none"> <li>this           <ul style="list-style-type: none"> <li>byteBuffer</li> <li>socketChannel</li> <li>result               <ul style="list-style-type: none"> <li>value</li> </ul> </li> <li>historyContext               <ul style="list-style-type: none"> <li>count</li> <li>value</li> </ul> </li> </ul> </li> </ul>	SocketChannelReadHandle (id=828) HeapByteBuffer (id=833) WindowsAsynchronousSocketChannelImpl (id=792) Integer (id=81) 50 StringBuffer (id=832) 0 (id=834)

图 7-26 客户端的对象情况 (2)

客户端 1: WindowsAsynchronousSocketChannelImpl: 760 | FileDescriptor: 762 | SocketChannelReadHandle: 803 | HeapByteBuffer: 808

客户端 2: WindowsAsynchronousSocketChannelImpl: 792 | FileDescriptor: 797 | SocketChannelReadHandle: 828 | HeapByteBuffer: 833

从以上的运行效果可以看到，服务器处理每一个客户端通道所使用的 **SocketChannelReadHandle**（处理器）对象都是独立的，并且所引用的 **SocketChannel** 对象也都是独立的。

**Java NIO 和 Java AIO 框架**，除了因为操作系统的实现不一样而去掉了 **Selector**，其他的重要概念都是相似的。例如上文中提到的 **Channel** 的概念，还有演示代码中使用的 **Buffer** 缓存方式。实际上 **Java NIO** 和 **Java AIO** 框架各位读者可以看成是一套完整的“高并发 I/O 处理”的实现。

### 7.5.3 还有改进可能

以上代码是示例代码，目标是让读者了解 **Java AIO** 框架的基本使用。所以它还有很多改造空间，例如：

- 在生产环境下，我们需要记录这个通道上“用户的登录信息”。那么这个需求可以使用 **Java AIO** 中的“附件”功能进行实现。
- 我们在以上章节为读者呈现的示例代码，都是使用“自定义文本”格式传输内容，并检查“over”关键字作为数据内容传输完成的标记。但是在正式生产环境下，建议使用更规范的数据格式。
- 我们可以使用 **json** 格式记录信息：因为它在相同的压缩率的前提下，有更好的信息结构；还可以使用 **protobuf** 组件中的信息描述方式：因为它兼顾传输效率和良好的信息结构；也还可以使用 **TLV** 格式，因为它提供很好的信息传输效率，等等。
- **Java AIO** 和 **Java NIO** 框架都支持线程池的使用，线程池的使用原则一定是只有业务处理部分才使用，使用后马上结束线程的执行。最好不要在其中再使用悲观锁或者乐观锁进行资源锁定（但在保证可以脱离阻塞状态的前提下，使用信号量工具在线程间进行控制是可以的）。**Java AIO** 框架中还有一个线程池，是拿给“通知处理器”使用的，因为 **Java AIO** 框架是基于“订阅——通知”模型的，所以“订阅”操作可以由主线程完成，但是不可能要求在应用程序中并发的“通知”操作也在主线程上完成。

## 7.6 第三方组件：Netty

**Netty** 是由 **JBOSS** 提供的一个 **Java** 开源框架。**Netty** 提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。在官网上对于 **Netty** 的介绍有这样一段文字：

Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server.

‘Quick and easy’ doesn’t mean that a resulting application will suffer from a

maintainability or a performance issue. Netty has been designed carefully with the experiences earned from the implementation of a lot of protocols such as FTP, SMTP, HTTP, and various binary and text-based legacy protocols. As a result, Netty has succeeded to find a way to achieve ease of development, performance, stability, and flexibility without a compromise.

实际上，Netty 框架并不只是封装了多路复用的 I/O 模型，也提供了对传统的阻塞式/非阻塞式同步模型的封装。当然，Netty 官网上的几段文字并不能概括 Netty 的全部作用。下面的内容将会在读者已经理解原生的 Java NIO 框架的基础上，介绍 Netty 的基本使用和工作原理。

这里说明一下，讲解如何使用 Netty 并不是本章的重点。关于 Netty 的使用介绍当然是官方文档 *JBOSS-Netty Architectural Overview* (<http://docs.jboss.org/netty/3.2/guide/html/>) 或者专题书籍上的内容更出色。本书讲解 Netty 的目的是，通过讲解 I/O 通信模型、Java 对各种通信模型的支持、上层的 Netty 封装，让大家深入理解“系统间通信”中的一个要素——信息如何传递。

### 7.6.1 为什么需要 Netty

有的读者可能会问，既然 Java NIO、Java AIO 框架已经实现了各主流操作系统的底层支持，那么为什么现在主流的 Java NIO 技术会是 Netty 和 MINA 呢？答案很简单：因为更好用！这里举几个方面的例子。

- 虽然 Java NIO 和 Java AIO 框架分别提供了“多路复用 I/O 和异步 I/O”的支持，但是并没有提供上层“消息格式”的良好封装。例如前两者并没有提供针对 Protocol Buffer、JSON 这些信息格式的封装，但是 Netty 框架提供了这些数据格式封装——基于责任链模式的编码和解码功能。
- 要编写一个可靠的、易维护的、高性能的（注意它们的排序）NIO/AIO 服务器应用。除了框架本身要兼容各类操作系统的实现，更重要的是它还要处理很多上层特有服务。例如客户端的权限，还有上面提到的信息格式封装、简单的数据读取。这些 Netty 框架都提供了相应的支持。
- Java NIO 框架存在一个“poll/epoll bug: Selector doesn't block on Selector.select (timeout)”的 Bug，不能阻塞意味着 CPU 的使用率会变成 100%（这是底层 JNI 的问题，其上层要处理这个异常实际上也不是难题）。另外这个 Bug 只有在 Linux 内核上才能重现。
- 以下这个问题在 JDK 1.7 版本中还没有被完全解决：[http://bugs.java.com/bugdatabase/view\\_bug.do?bug\\_id=2147719](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2147719)。虽然 Netty 4.0 也是基于 Java NIO 框架进行封装的（上文中已经给出了 Netty 中 NIOServerSocketChannel 类的介绍），但是 Netty 已经将这个 Bug 在上一层进行了处理。

## 7.6.2 Netty 快速上手

Netty 的使用非常简单，以下这段代码是一个 Netty 的基本使用示例。代码本身比较好理解，本书在代码中又加上了比较详细的注解，相信就算读者之前没有使用过 Netty，也应该可以看懂。如果读者之前接触过 Netty，那么可以发现，这段代码中基本上已经包含了 Netty 比较重要的几个概念了：Channel、Buffer、ChannelPipeline、ChannelHandler、ChannelHandlerContext 等。

### • Netty For Server 端代码

```
package testNetty;
.....

import io.netty.bootstrap.ServerBootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.ChannelHandler.Sharable;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.bytes.ByteArrayDecoder;
.....

public class TestTCPNetty {
    public static void main(String[] args) throws Exception {
        //这就是主要的服务启动器
        ServerBootstrap serverBootstrap = new ServerBootstrap();
        //=====下面设置线程池
        //Boss 线程池
        EventLoopGroup bossLoopGroup = new NioEventLoopGroup(1);
        //Work 线程池：这样的申明方式，主要是为了向读者说明 Netty 的线程组是怎样工作的
        ThreadFactory threadFactory = new DefaultThreadFactory("work
thread pool");
        //CPU 个数
        int processorsNumber = Runtime.getRuntime().availableProcessors();
        EventLoopGroup workLoogGroup =
        //指定 Netty 的 Boss 线程和 Work 线程
        serverBootstrap.group(bossLoopGroup, workLoogGroup);
        //如果是以下的申明方式，则说明 Boss 线程和 Work 线程共享一个线程池
        //serverBootstrap.group(workLoogGroup);
        //=====下面我们设置服务的通道类型
```



```

//只能是实现了 ServerChannel 接口的“服务器”通道类
serverBootstrap.channel(NioServerSocketChannel.class);
//当然也可以这样创建 (SelectorProvider 是不是感觉很熟悉)
//serverBootstrap.channelFactory(new
ChannelFactory<NioServerSocketChannel>() {
    // @Override
    // public NioServerSocketChannel newChannel() {
    // return new NioServerSocketChannel(SelectorProvider.provider
    ());
    // }
    //});
//=====设置处理器
//为了演示, 这里设置了一组简单的 ByteArrayDecoder 和 ByteArrayEncoder
//Netty 的特色就在这一连串“通道水管”中的“处理器”
serverBootstrap.childHandler(new
ChannelInitializer<NioSocketChannel>() {
    @Override
    protected void initChannel(NioSocketChannel ch) throws
Exception {
        ch.pipeline().addLast(new ByteArrayEncoder());
        ch.pipeline().addLast(new TCPServerHandler());
        ch.pipeline().addLast(new ByteArrayDecoder());
    }
});
//=====设置 Netty 服务器绑定的 IP 和端口
serverBootstrap.option(ChannelOption.SO_BACKLOG, 128);
serverBootstrap.childOption(ChannelOption.SO_KEEPALIVE, true);
serverBootstrap.bind(new InetSocketAddress("0.0.0.0", 83));
//还可以监控多个端口
//serverBootstrap.bind(new InetSocketAddress("0.0.0.0", 84));
}
}
@Sharable
class TCPServerHandler extends ChannelInboundHandlerAdapter {
    private static Log LOGGER = LoggerFactory.getLog(TCPServerHandler.
class);
    // 每一个 Channel, 都有独立的 handler、ChannelHandlerContext、
    // ChannelPipeline、Attribute
    // 所以不需要担心多个 Channel 中的这些对象相互影响
    // 这里我们使用 Content 这个 Key, 记录这个 handler 中已经接收到的客户端信息

    private static AttributeKey<StringBuffer> content = AttributeKey.
valueOf("content");
    @Override

```

```

        public void channelRegistered(ChannelHandlerContext ctx) throws
Exception {
            TCPServerHandler.LOGGER.info("super.channelRegistered(ctx)");
        }
        @Override
        public void channelUnregistered(ChannelHandlerContext ctx) throws
Exception{
            TCPServerHandler.LOGGER.info("super.channelUnregistered(ctx)");
        }
        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception
        {
            TCPServerHandler.LOGGER.info("super.channelActive(ctx) = " + ctx.
toString());
        }
        @Override
        public void channelInactive(ChannelHandlerContext ctx) throws
Exception {
            TCPServerHandler.LOGGER.info("super.channelInactive(ctx)");
        }
        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
            TCPServerHandler.LOGGER.info("channelRead(ChannelHandlerContext
ctx, Object msg)");
            //我们使用 IDE 工具模拟长连接中的数据缓慢提交
            //由 read 方法负责接收数据, 但只是进行数据累加, 不进行任何处理
            ByteBuf byteBuf = (ByteBuf)msg;
            try {
                StringBuffer contextBuffer = new StringBuffer();
                while(byteBuf.isReadable()) {
                    contextBuffer.append((char)byteBuf.readByte());
                }
                //加入临时区域
                StringBuffercontent= ctx.attr(TCPServerHandler. content).get
                ();
                if(content == null) {
                    content = new StringBuffer();
                    ctx.attr(TCPServerHandler.content).set(content);
                }
                content.append(contextBuffer);
            } catch(Exception e) {
                throw e;
            } finally {

```

```

        byteBuf.release();
    }
}
@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
    TCPServerHandler.LOGGER.info("super.channelReadComplete(ChannelHandlerContext ctx)");
    // 由 readComplete 方法负责检查数据是否接收完了
    // 和本章 7.3~7.5 节中示例代码的读取规定相同
    StringBuffer content = ctx.attr(TCPServerHandler.content).get();
    //如果条件成立，则说明还没有接收到完整的客户端信息
    if(content.indexOf("over") == -1) {
        return;
    }
    //当接收到信息后，首先要做的是清空原来的历史信息
    ctx.attr(TCPServerHandler.content).set(new StringBuffer());
    //准备向客户端发送响应
    ByteBuf byteBuf = ctx.alloc().buffer(1024);
    byteBuf.writeBytes("回发响应信息!".getBytes());
    ctx.writeAndFlush(byteBuf);
    //正常终止这个通道上下文，就可以关闭通道了
    //如果不关闭，这个通道的会话将一直存在，
    //只要网络是稳定的，服务器就可以随时通过这个会话向客户端发送信息
    //关闭通道意味着 TCP 将正常断开，其中所有的
    //handler、ChannelHandlerContext、ChannelPipeline、Attribute 等信息
    //都将被注销
    ctx.close();
}
}

```

以上示例代码工作在 Server 端，客户端是否使用了 NIO 技术实际上对整个软件架构的性能影响不大。你可以使用任何支持 TCP/IP 协议技术的代码作为客户端，无论是 Python、C++、C#还是 Java。

- 以下代码是客户端代码，它在之前的内容中已经出现过，这里再赘述一次（不加注释）：

```

package testBSocket;

public class SocketClientRequestThread implements Runnable {
    private static final Log LOGGER = LogFactory.getLog(SocketClientRequestThread.class);
    private CountdownLatch countDownLatch;

```



```

        private Integer clientIndex;
        public SocketClientRequestThread(CountDownLatch countDownLatch ,
Integer clientIndex) {
            this.countDownLatch = countDownLatch;
            this.clientIndex = clientIndex;
        }
        @Override
        public void run() {
            Socket socket = null;
            OutputStream clientRequest = null;
            InputStream clientResponse = null;
            try {
                socket = new Socket("localhost",83);
                clientRequest = socket.getOutputStream();
                clientResponse = socket.getInputStream();
                this.countDownLatch.await();
                clientRequest.write(URLEncoder.encode("第" + this.clientIndex
+ " 个客户端请求 11.", "UTF-8").getBytes());
                clientRequest.flush();
                clientRequest.write(URLEncoder.encode("第" + this.clientIndex
+ " 个客户端请求 22. over", "UTF-8").getBytes());
                SocketClientRequestThread.LOGGER.info("第" + this.clientIndex
+ " 个客户端请求发送完成，等待服务器返回");
                int maxLen = 1024;
                byte[] contextBytes = new byte[maxLen];
                int realLen;
                String message = "";
                while((realLen=clientResponse.read(contextBytes, 0, maxLen))!= -
1) {
                    message += new String(contextBytes , 0 , realLen);
                }
                SocketClientRequestThread.LOGGER.info("接收到来自服务器的信息:"
+ message);
            } catch (Exception e) {
                SocketClientRequestThread.LOGGER.error(e.getMessage(), e);
            } finally {
            }
        }
    }
}

```

虽然示例代码（前文中）中已经有比较详细的注释说明，但是为了让读者更清楚服务器代码的含义，下面我们将针对代码中重要的内容进行讲解。

```
//Boss 使用的线程池
```

```
EventLoopGroup bossLoopGroup = new NioEventLoopGroup(1);
```

Boss 线程池实际上就是 Java NIO 框架中的 Selector 工作角色，针对一个本地 IP 的端口，Boss 线程池中有一条线程工作，工作内容也相对简单，就是发现新的连接；Netty 支持同时监听多个端口，所以 Boss 线程池的大小按照需要监听的服务器端口数量进行设置。

```
//Work 线程池
int processorsNumber = Runtime.getRuntime().availableProcessors();
EventLoopGroup workLoogGroup = new NioEventLoopGroup(processorsNumber *
2, threadFactory, SelectorProvider.provider());
```

这段代码主要是确定 Netty 中工作线程池的大小，这个大小一般是物理机器/虚拟机器可用内核的个数乘以 2。Work 线程池中的线程（如果封装的是 Java NIO，那么具体的线程实现类就是 NIOEventLoop）都固定负责指派给它的网络连接的事件监听，并根据状态调用不同的 ChannelHandler 事件方法。而最后一个参数 SelectorProvider 说明了这个 EventLoop 所使用的多路复用 I/O 模型的具体实现由操作系统决定。

```
serverBootstrap.option(ChannelOption.SO_BACKLOG, 128);
serverBootstrap.childOption(ChannelOption.SO_KEEPALIVE, true);
```

option 方法可以设置这个 ServerChannel 相应的各种属性（在代码中我们使用的是 NIOServerSocketChannel）；childOption 方法用于设置这个 ServerChannel 收到客户端事件后，所生成的新的 Channel 的各种属性（代码中，我们生成的是 NIOSocketChannel）。更详细的 option 参数可以参见 ChannelOptiOn 类中的注释说明。

### 7.6.3 Netty 中的重要概念

#### 1. Netty 线程机制

还记得在讲解 Java NIO 框架对多路复用 I/O 模型的支持时讲到的 Selector（选择器）吗？它大致的工作方式是：

```
while(true) {
    if(selector.select(100) == 0) {
        continue;
    }
    Iterator<SelectionKey> selecionKeys = selector.selectedKeys().
iterator();
    while(selecionKeys.hasNext()) {
        SelectionKey readyKey = selecionKeys.next();
        selecionKeys.remove();
        SelectableChannel selectableChannel = readyKey.channel();
        if(readyKey.isValid() && readyKey.isAcceptable()) {
            .....
        } else if(readyKey.isValid()&&readyKey.isConnectable()) {
```



```

        .....
    } else if(readyKey.isValid() && readyKey.isReadable()) {
        .....
    }
}
}

```

Selector 可以在主线程上面操作，也可以在一个独立的线程上进行操作。在 Netty 中，这里的部分工作就交给 Boss 线程完成，而且建议读者使用线程池技术。Boss 线程负责发现连接到服务器的新的 Channel (SocketServerChannel 的 ACCEPT 事件)，并且将这个 Channel 经过检查后注册到 Work 连接池的某个 EventLoop 线程中。而当 Work 线程发现操作系统有一个它感兴趣的 I/O 事件时 (例如 SocketChannel 的 READ 事件)，则调用相应的 ChannelHandler 事件。当某个 Channel 失效后 (例如显示调用 ctx.close())，这个 Channel 将从绑定的 EventLoop 中被剔除。

在 Netty 中，如果我们使用的是 Java NIO 框架实现的对多路复用 I/O 模型的支持，那么进行这个循环的是 NIOEventLoop 类 (可参见该类中的 processSelectedKeysPlain 方法和 processSelectedKey 方法)。另外在这个类中 Netty 解决了之前说到的 Java NIO 中 “Selector.select(timeout) CPU 100%” 的 Bug 和一个 “NullPointerException in Selector.open()” 的 Bug:

```

// processSelectedKeysPlain 方法片段
for (;;) {
    final SelectionKey k = i.next();
    final Object a = k.attachment();
    i.remove();
    if (a instanceof AbstractNioChannel) {
        processSelectedKey(k, (AbstractNioChannel) a);
    } else {
        @SuppressWarnings("unchecked")
        NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
        processSelectedKey(k, task);
    }
    if (!i.hasNext()) {
        break;
    }
    if (needsToSelectAgain) {
        selectAgain();
        selectedKeys = selector.selectedKeys();
        // Create the iterator again to avoid ConcurrentModification
        // Exception
        if (selectedKeys.isEmpty()) {
            break;
        } else {

```

```

        i = selectedKeys.iterator();
    }
}
// processSelectedKey 方法片段
if (!k.isValid()) {
    // close the channel if the key is not valid anymore
    unsafe.close(unsafe.voidPromise());
    return;
}
try {
    int readyOps = k.readyOps();
    // Also check for readOps of 0 to workaround possible JDK bug which
    // may otherwise lead
    // to a spin loop
    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) !=
0 || readyOps == 0) {
        unsafe.read();
        if (!ch.isOpen()) {
            // Connection already closed - no need to handle write.
            return;
        }
    }
    if ((readyOps & SelectionKey.OP_WRITE) != 0) {
        // Call forceFlush which will also take care of clear the
        // OP_WRITE once there is nothing left to write
        ch.unsafe().forceFlush();
    }
    if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
        // remove OP_CONNECT as otherwise Selector.select(..) will
        // always return without blocking
        // See https://github.com/netty/netty/issues/924
        int ops = k.interestOps();
        ops &= ~SelectionKey.OP_CONNECT;
        k.interestOps(ops);
        unsafe.finishConnect();
    }
} catch (CancelledKeyException ignored) {
    unsafe.close(unsafe.voidPromise());
}
}

```

一个 Work 线程池的线程将按照底层封装 Java NIO 框架中 Selector 的事件状态，决定执行 ChannelHandler 中的哪一个事件方法（Netty 中包括了 channelRegistered、channelUnregistered、channelActive、channelInactive 等事件方法）。执行完成后，Work 线程将一直轮询直到操作系统回复下一个它所管理的 Channel 发生了新的 I/O 事件。

## 2. ByteBuf

Netty uses its own buffer API instead of Nio ByteBuffer to represent a sequence of bytes. This approach has significant advantages over using ByteBuffer. Netty's new buffer type, ChannelBuffer has been designed from the ground up to address the problems of ByteBuffer and to meet the daily needs of network application developers. To list a few cool features:

- You can define your own buffer type if necessary.
- Transparent zero copy is achieved by a built-in composite buffer type.
- A dynamic buffer type is provided out-of-the-box, whose capacity is expanded on demand, just like StringBuffer.
- There's no need to call flip() anymore.
- It is often faster than ByteBuffer.

上面的引用来自于 JBOSS-Netty 官方文档中对 ByteBuf 缓存的解释。翻译成中文大致的意思就是：Netty 重写了 Java NIO 框架中的缓存结构，并将这个结构应用在更上层的封装中。为什么要重写呢？JBOSS-Netty 给出的解释是：这个缓存比 Java 中的 ByteBuffer 牛！

图 7-27 是 Netty 中继承自上层 ByteBuf 的部分类结构。

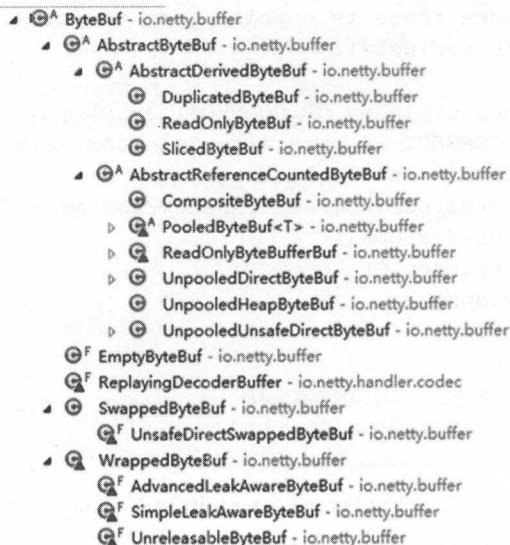


图 7-27 Netty-ByteBuf 继承结构

这里说一说 Netty 中几个比较特别的 ByteBuf 实现。

- **io.netty.buffer.EmptyByteBuf**: 这是一个初始容量和最大容量都为 0 的缓存区。一般我们用这种缓存区描述“没有任何处理结果”，并将其向下一个 Handler 传递。
- **io.netty.buffer.ReadOnlyByteBuf**: 这是一个不允许任何“写请求”的只读缓存区。一般通过 `Unpooled.unmodifiableBuffer(ByteBuf)` 方法将某一个可正常读写的缓存区转变而成。如果我们需要在下一个 Handler 处理的过程中禁止写入任何数据到缓存区，那么就可以在这个 Handler 中进行“只读缓存区”的转换。
- **io.netty.buffer.UnpooledDirectByteBuf**: 基本的 Java NIO 框架的 ByteBuffer 封装。一般直接使用这个缓存区实现来处理 Handler 事件。
- **io.netty.buffer.PooledByteBuf**: Netty 4.x 版本的缓存新特性，主要是为了减少之前 `unpoolByteBuf` 在创建和销毁时的 GC 时间。

### 3. Channel

Channel 可译为通道。你可以使用 Java NIO 中的 Channel 去初次理解它，但实际上它的意义和 Java NIO 中的通道意义还不一样。我们可以解释成：“更抽象、更丰富”，如图 7-28 所示。

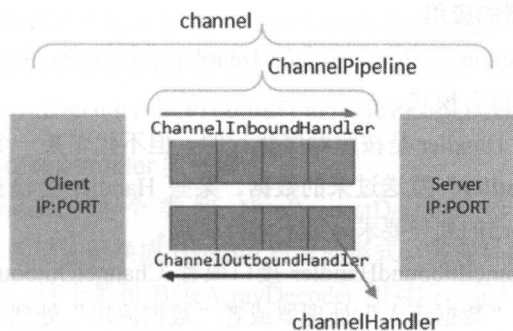


图 7-28 Netty 中的 Channel

- Netty 中的 Channel 专门代表网络通信，这个和 Java NIO 框架中的 Channel 不一样，后者中还有类似 `FileChannel` 本地文件的 I/O 通道。由于前者专门代表网络通信，所以它是由客户端地址 + 服务器地址 + 网络操作状态构成的，请参见 `io.netty.channel.Channel` 接口的定义。
- 每一个 Netty 中的 Channel 比 Java NIO 中的 Channel 更抽象。这是为什么呢？在 Netty 中，不止封装了多路复用 I/O 模型，还封装了 Java BIO 支持的同步网络 I/O 通信模型。将它们的表现上都抽象成 Channel 了。而我们知道在 Java BIO 支持的同步网络 I/O 模型中，原来是不存在 Channel 这个概念的。

#### 4. ChannelPipeline 和 ChannelHandler

Netty 中的每一个 Channel，都有一个独立的 ChannelPipeline，中文名称为“通道水管”。只不过这个水管是双向的，里面流淌着数据，数据可以通过这个“水管”流入到服务器，也可以通过这个“水管”从服务器流出。

在 ChannelPipeline 中，有若干个过滤器，我们称之为“ChannelHandler”（也可以称为过滤器）。同“流入”和“流出”的概念相对应：用于处理/过滤“流入数据”的 ChannelHandler，被称为“ChannelInboundHandler”；用于处理/过滤“流出数据”的 ChannelHandler，被称为“ChannelOutboundHandler”，如图 7-29 所示。



图 7-29 ChannelHandler

##### (1) 责任链和适配器的应用

- 数据在 ChannelPipeline 中由一个的一个的 Handler 进行处理，并形成一个新的数据状态。这是典型的“责任链”模式。
- 虽然数据管道中的 Handler 是按照顺序执行的，但不代表某一个 Handler 会处理任何一种由“上一个 Handler”发送过来的数据。某些 Handler 会检查传来的数据是否符合要求，如果不符合自己的处理要求，则不进行处理。
- 我们可以实现 ChannelInboundHandler 接口或者 ChannelOutboundHandler 接口中的方法，来运行自己业务的“数据流入”处理器或者“数据流出”处理器。
- 但是这两个接口需要实现的事件方法是比较多的，例如 ChannelInboundHandler 接口一共有 11 个需要实现的接口方法（包括父级 ChannelHandler 的，在后面讲解 Channel 的生命周期时，会专门讲到这些事件的执行顺序和执行状态），一般情况下不需要把这些方法全部实现，因为我们不一定会关心所有这些事件，只会将精力放在某一个或者某几个事件方法上。

所以 Netty 中增加了两个适配器“ChannelInboundHandlerAdapter”和“ChannelOutboundHandlerAdapter”来帮助我们实现关心的事件方法，如图 7-30 所示。



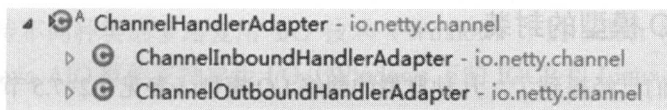


图 7-30 ChannelHandlerAdapter

在本书上文给出的示例代码中，书写的业务处理器 `TCPServerHandler` 就继承了 `ChannelInboundHandlerAdapter` 适配器。下面，我们将介绍几个常用的 `ChannelInboundHandler` 处理器和 `ChannelOutboundHandler` 处理器。

### (2) ChannelInboundHandler 类举例

- `HttpRequestDecoder`：实现了 HTTP 协议的数据输入格式的解析。这个类将数据编码为 `HttpMessage` 对象，并交由下一个 `ChannelHandler` 进行处理。
- `ByteArrayDecoder`：最基础的数据流输入处理器，将所有的 `Byte` 转换为 `ByteBuf` 对象（一般的实现类是 `io.netty.buffer.UnpooledUnsafeDirectByteBuf`）。我们进行文本格式信息传输到服务器时，最好使用 `Handler` 将 `Byte` 数组转换为 `ByteBuf` 对象。
- `DelimiterBasedFrameDecoder`：这个数据流输入处理器，会按照外部传入的数据中给定的某个关键字/关键字串重新将数据组装为新的段落并发送给下一个 `Handler` 处理器。后文中，我们将使用这个处理器进行 TCP 半包问题的处理。
- 还有很多直接支持标准数据格式解析的处理器，例如支持 Google Protocol Buffers 数据格式解析的 `ProtobufDecoder` 和 `ProtobufVarint32FrameDecoder` 处理器。

### (3) ChannelOutboundHandler 类举例

- `HttpResponseEncoder`：这个类和 `HttpRequestDecoder` 相对应，是将服务器端 `HttpReponse` 对象的描述转换成 `ByteBuf` 对象的形式，并向外传播。
- `ByteArrayEncoder`：这个类和 `ByteArrayDecoder` 相对应，是将服务器端的 `ByteBuf` 对象转换成 `Byte` 数组的形式，并向外传播。
- 还支持标准的编码成 Google Protocol Buffers 格式、JBoss Marshalling 格式、ZIP 压缩格式的 `ProtobufEncoder`、`ProtobufVarint32LengthFieldPrepender`、`MarshallingEncoder`、`JZlibEncoder` 等。

## 7.7 再次审视 Netty 的作用

Netty 框架的特性，使我们不需要关心下层逻辑所工作的网络 I/O 模型；利用 Netty 提供的面向事件驱动的方法结构，使我们更能集中精力关注应用层业务。那么本段内容将再次回过头审视“为什么要使用 Netty 这样的 NIO 业务框架”这个问题。

### 7.7.1 对网络 I/O 模型的封装

在本书前面我们已经提到了，几种典型的网络 I/O 模型（参见 7.2~7.5 节）。

- 阻塞和非阻塞：这个概念是针对应用程序而言的，是指应用程序中的线程在向操作系统发送 I/O 请求后，是否一直等待操作系统的 I/O 响应。如果是，那么就是阻塞式的；如果不是，那么应用程序一般会以轮询的方式以一定周期询问操作系统，直到某次获得了 I/O 响应为止（轮询间隔应用程序线程可以做一些其他工作）。
- 同步和异步：I/O 操作都是由操作系统进行的（这里的 I/O 操作涵盖了非常多的概念，磁盘 I/O、网络 I/O 都算），不同的操作系统对不同设备的 I/O 操作都有不同的模式。同步和异步这两个概念都是操作系统级别的，同步 I/O 是指操作系统和设备进行交互时，必须等待一次完整的“请求—响应”完成，才能进行下一次操作（当然操作系统和设备本身也有很多技术加快这个反应过程，例如“磁盘预读”技术、数据缓存技术）；异步 I/O 是指操作系统和设备进行交互时，不必等待本次得到响应，就可以直接进行下一次操作请求，设备处理完某次请求后，会主动给操作系统相应的响应通知。
- 多路复用 I/O：多路复用 I/O，从本质上看还是一种同步 I/O，因为它没有百分之百消除 IO\_WAIT 状态，操作系统也没有为它提供“主动通知”机制。但是多路复用 I/O 的处理速度已经相当快了，利用设备执行 I/O 操作的时间，操作系统可以继续执行 I/O 请求，并同样采用周期性轮询的方式，获取一批 I/O 操作请求的执行响应。各种不同的操作系统支持的多路复用 I/O 模型包括有 select、poll、epoll、kqueue 等。
- 阻塞式同步 I/O 模型：这个从字面上就很好理解，应用程序请求 I/O 操作，并一直等待处理结果；操作系统同时也进行 I/O 操作，并等待设备的处理结果；可以看出，应用程序的请求线程和操作系统的内核线程都是等待状态。
- 非阻塞式同步 I/O 模型：应用程序请求 I/O，并且不用一直等待返回结果就去做其他事情。间隔一定的时间周期，再去询问操作系统上次 I/O 操作有没有结果，直到某一次询问从操作系统拿到 I/O 结果；操作系统内核线程在进行 I/O 操作时，还是处于一直等待设备返回 I/O 操作结果的状态。
- 多路复用 I/O 模型：应用程序请求 I/O 的工作采用非阻塞方式进行，操作系统采用多路复用模式工作。
- 非阻塞式异步 I/O 模型：应用程序请求 I/O 的工作采用非阻塞方式进行，但是不需要轮询了，因为操作系统异步 I/O 其中一个重要特性就是，可以在有 I/O 响应结果的时候，主动通知上层进程。

以上这些 I/O 工作模型，在 Java 中都能够找到对应的支持：传统的 Java Socket 套接字支持阻塞/非阻塞模式下的同步 I/O（有的技术资料里面也称为 OIO 或者 BIO）；Java NIO 框架在

不同操作系统下支持不同种类的多路复用 I/O 技术（Windows 下的 select 模型、Linux 下的 poll/epoll 模型）；Java AIO 框架支持异步 I/O（Windows 下的 IOCP 和 Linux 使用 epoll 的模拟 AIO）。

实际上 Netty 是对 Java BIO、Java NIO 框架的再次封装。让我们不再纠结于选用哪种底层实现。读者可以理解成 **Netty 框架** 是一个面向上层业务实现进行封装的“业务层”框架。而 **Java Socket 框架、Java NIO 框架、Java AIO 框架** 更偏向于对下层技术实现的封装，是面向“技术层”的框架。

### 7.7.2 对数据信息格式的封装

“技术层”框架本身只对 I/O 模型实现进行了封装，并不关心 I/O 模型中流淌的数据格式；“业务层”框架对数据格式也进行了处理，让我们可以抽出精力关注业务本身。

- **Protobuf 数据协议的集成**：Netty 利用自身的 ChannelPipeline 的设计，对 Protobuf 数据协议进行了无缝结合。
- **JBoss Marshalling 数据协议的集成**：JBoss Marshalling 是一个 Java 对象的序列化 API 包，修正了 JDK 自带的序列化包的很多问题，又保持了跟 java.io.Serializable 接口的兼容。Netty 通过封装这个协议，可以帮助我们在“客户端—服务器端”简便地进行对象序列化和反序列化。
- **HTTP Request / HTTP Response 协议的集成**：在 Netty 中，可以方便地接收和发送 HTTP 协议。也就是说，我们可以使用 Netty 搭建自己的 Web 服务器，当然还可以根据自己的业务要求，方便地设计出类似于 Struts、Spring MVC 这样的 Web 框架。

下面是一个使用 Netty 的 HTTP 编码/解码处理器设计的一个简单的 Web 服务器：

```
package testNetty;
.....
public class TestHTTPNetty {
    .....
    public static void main(String[] args) throws Exception {
        //这就是主要的服务启动器
        ServerBootstrap serverBootstrap = new ServerBootstrap();
        //=====下面设置线程池
        EventLoopGroup bossLoopGroup = new NioEventLoopGroup(1);
        ThreadFactory threadFactory = new DefaultThreadFactory("work
thread pool");
        int processorsNumber = Runtime.getRuntime().availableProcessors();
        EventLoop GroupworkLoogGroup = newNioEventLoopGroup (processors
Number* 2, threadFactory, SelectorProvider.provider());
        serverBootstrap.group(bossLoopGroup , workLoogGroup);
```

```
//=====下面设置服务的通道类型（代码已经详细讲解过，就不再赘述了）
serverBootstrap.channel(NioServerSocketChannel.class);
//=====设置处理器
serverBootstrap.childHandler(new
ChannelInitializer<NioSocketChannel>() {
    @Override
    protected void initChannel(NioSocketChannel ch) throws
Exception {
        //我们在 socket channel pipeline 中加入 HTTP 的编码/解码器
        ch.pipeline().addLast(new HttpResponseEncoder());
        ch.pipeline().addLast(new HttpRequestDecoder());
        ch.pipeline().addLast(new HTTPServerHandler());
    }
});
serverBootstrap.option(ChannelOption.SO_BACKLOG, 128);
serverBootstrap.childOption(ChannelOption.SO_KEEPALIVE, true);
serverBootstrap.bind(new InetSocketAddress("0.0.0.0", 83));
}
}
@Sharable
class HTTPServerHandler extends ChannelInboundHandlerAdapter {
    private static Log LOGGER = LoggerFactory.getLog(HTTPServerHandler.
class);
    //由于一次 HttpContent 可能没有传输完全部的请求信息，所以这里要做一个连续的记录
    //然后在 channelReadComplete 方法中（执行了这个方法说明这次所有的 HTTP 内容都传
    //输完了）进行处理
    private static AttributeKey<StringBuffer> CONTENT = AttributeKey.
valueOf("content");
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        /*
        * 在测试中，首先取出客户端传来的参数、URL 信息，并且返回一个确认信息
        * 要使用 HTTP 服务，我们首先要了解 Netty 中 HTTP 的格式，如下：
        * -----
        * | http request | http content | http content |
        * -----
        * 所以通过 HttpRequestDecoder channel handler 解码后的 msg 可能有两种
        * 类型：
        * HttpRequest: 里面包含了请求 head、请求的 URL 等信息
        * HttpContent: 请求的主体内容
        * */
        if(msg instanceof HttpRequest) {
            HttpRequest request = (HttpRequest)msg;
```



```

        HttpMethod method = request.getMethod();
        String methodName = method.name();
        String url = request.getUri();
        HTTPServerHandler.LOGGER.info("methodName = " + methodName +
" && url = " + url);
    }
    //如果条件成立，则在这个代码段实现 HTTP 请求内容的累加
    if(msg instanceof HttpContent) {
        StringBuffer content = ctx.attr(HTTPServerHandler.
CONTENT).get();
        if(content == null) {
            content = new StringBuffer();
            ctx.attr(HTTPServerHandler.CONTENT).set(content);
        }
        HttpContent httpContent = (HttpContent)msg;
        ByteBuf contentBuf = httpContent.content();
        String preContent = contentBuf.toString(io.netty.util.
CharsetUtil.UTF_8);
        content.append(preContent);
    }
}
@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
    HTTPServerHandler.LOGGER.info("super.channelReadComplete(ChannelHandlerConte
xt ctx)");
    //一旦本次 HTTP 请求传输完成，就可以进行业务处理了
    //并且返回响应
    StringBuffer content = ctx.attr(HTTPServerHandler.CONTENT).get
();
    HTTPServerHandler.LOGGER.info("HTTP 客户端传来的信息为：" + content);
    //开始返回信息了
    String returnValue = "return response";
    FullHttpResponse response = new DefaultFullHttpResponse (HttpVersion.
HTTP_1_1, HttpResponseStatus.OK);
    HttpHeaders httpHeaders = response.headers();
    //这些就是 HTTP response 的 head 信息，参见 HTTP 规范。另外还可以设置自己的
    //head 属性
    httpHeaders.add("param", "value");
    response.headers().set(HttpHeaders.Names.CONTENT_TYPE,
"text/plain");
    //一定要设置长度，否则 HTTP 客户端会一直等待（因为返回的信息长度客户端不知道）
    response.headers().set(HttpHeaders.Names.CONTENT_LENGTH,

```



```
returnValue.length());  
    ByteBuf responseContent = response.content();  
    responseContent.writeBytes(returnValue.getBytes("UTF-8"));  
    //开始返回  
    ctx.writeAndFlush(response);  
}  
}
```

由于本节的其他段落已经介绍了 Netty 的基本使用方法，所以以上的代码将其他不必要的注释、方法都去掉了，只做了实现 Web 服务器的最简代码。但是这段代码是可以运行的。如图 7-31 所示是运行效果（PostMan 测试效果）。

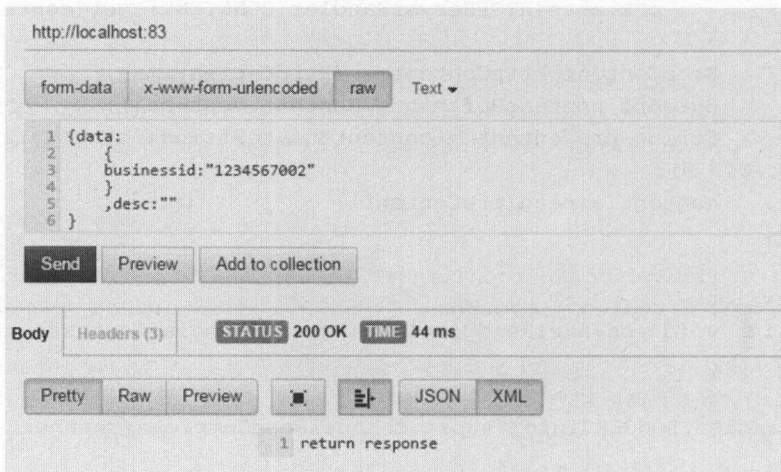


图 7-31 HTTP 消息格式封装实例运行效果

### 7.7.3 解决了“技术层”框架中的技术问题

通过阅读 Netty 框架的代码，我们知道了 Netty 框架至少解决了 Java NIO 框架中的一些 Bug。例如编号为 JDK-6427854 的 Bug：(se) NullPointerException in Selector.open()。http://bugs.java.com/view\_bug.do?bug\_id=6427854。这个 Bug 的官方描述是：

sun.nio.ch.Util contains code which is not thread safe and can throw a  
NullPointerException:

```
private static String bugLevel = null;  
static boolean atBugLevel(String bl) { // package-private  
    if (bugLevel == null) {  
        if (!sun.misc.VM.isBooted())
```

```

return false;
java.security.PrivilegedAction pa =
new GetPropertyAction("sun.nio.ch.bugLevel");
// the next line can reset bugLevel to null
bugLevel = (String)AccessController.doPrivileged(pa);
if (bugLevel == null)
bugLevel = "";
}
return (bugLevel != null) && bugLevel.equals(bl);
}

```

Suppose that two threads enter the "if (buglevel == null)" body at the same time. The first one runs until the return line and gets scheduled out right after the (buglevel != null) check. The second one then runs until right after the doPrivileged() call, sets bugLevel to null and gets scheduled out. The first one continues and hits a NullPointerException while calling bugLevel.equals() with bugLevel being null.

这个问题在 Netty 框架中，在负责进行 Java NIO Selector 的 NIOEventLoop 类中得到了解决。

再例如另一个 Bug: workaround the infamous epoll 100% CPU bug ([http://bugs.java.com/bugdatabase/view\\_bug.do?bug\\_id=6403933](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933)) 这个 Bug 出现在 Linux 系统环境，大致是说 Java NIO 框架在实现 Linux 内核 kernel 2.6+ 中的 epoll 模型时，Selector.select(timeout) 方法不能阻塞指定的 timeout 时间，导致 CPU 100% 的情况：

#### A DESCRIPTION OF THE PROBLEM :

Trying to get all bindings from the transient nameserver brings orbd into a state where it consumes 100% CPU. Its interesting to note that the problem only occurs if the client is programmed in c++. I was not able to reproduce the problem with a client programmed in Java.

#### STEPS TO FOLLOW TO REPRODUCE THE PROBLEM :

- 1) Get omniORB 4.1 from <http://omniORB.sourceforge.net>
  - 2) Compile omniORB (requires python devel package installed)
- ```

cd /tmp
mkdir omni_local
tar xvzf omniORB-4.1.0.tar.gz

```

```

cd omniorB-4.1.0
./configure --prefix=/tmp/omni_local
make
make install
3) Compile the test program (binding_browser, source attached to this report)
g++ -I/tmp/omni_local/include -L/tmp/omni_local/lib -lomniorB4 -lomniDynamic4 -
lomnithread -lpthread -lrt BindingBrowser.cc -o binding_browser
4) Start orbd
<JAVA_HOME>/bin/orbd -ORBInitialPort 12345
5) Start binding_browser (from another shell)
5a) export LD_LIBRARY_PATH=/tmp/omni_local/lib:$LD_LIBRARY_PATH
5b) ./binding_browser -ORBInitRef NameService=corbaloc::1.2@localhost:12345/
TNameService
Repeat step 5b until orbd consumes 100% cpu.

```

这个问题从官方的 Bug Database 中的描述看，是在 JDK 7 版本中被解决的。Netty 框架在 JDK 6+ 的环境下在 Java NIO 框架封装之上解决了这个 Bug。

#### 7.7.4 解决半包问题和粘包问题

我们考虑一下这样的情况：我们编写了一个机器人控制程序，通过一个遥控器（客户端）向机器人（服务器）建立了一个长连接，并通过这个连接连续不断地从遥控器发送控制指令给机器人。由于是连续控制指令，所以指令与指令之间没有间隔（实际上你还可以想想很多类似场景，例如多人网络在线对战游戏）。

我们使用 JSON 格式作为指令数据的承载格式。那么发送方和接收方的数据“发送—接收”过程可能如图 7-32 所示。

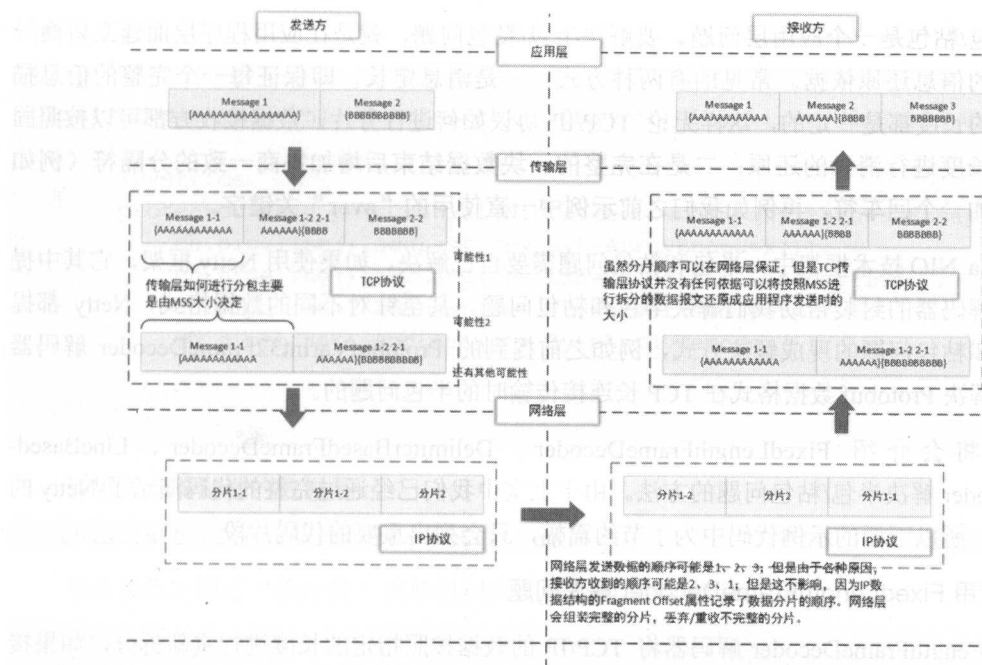


图 7-32 粘包问题

通过图 7-32 我们看到了接收方为了接收这两条连贯的指令，一共做了三次接收。第二次接收的时候，收到了一部分 Message1 的内容和一部分 Message2 的内容。这里要说明几个注意事项。

- **MSS:** MSS 属性是 TCP 连接双方在三次握手时所确认的每一个 TCP 报文段中数据字段的最大长度。注意两个问题：第一个是连接双方协商出来的；第二个是，这个数据只是数据段的最大长度，不包括 IP 协议头和 TCP 协议头的最大长度。
- **半包是指，**接收方应用程序在接收信息时，没有接收到一个完整的信息格式块；**粘包是指，**接收方应用程序在接收信息时，除了接收到发送方应用程序发送的某一个完整数据信息描述，还接收到了一下发送方应用程序发送的下一个数据信息的一部分。
- **半包和粘包是针对应用程序来说的，**这个问题只会发生在 TCP 协议进行连续发送数据时（TCP 长连接）。UDP 不会出现这个问题，因为 UDP 都是有边界的数据报；TCP 短连接也不会出现，因为发送完一个指令信息后连接就断开了，不会发送第二个指令数据。
- **半包和粘包问题产生的根本是因为 TCP 本质上没有“数据块”的概念，而是一连串的数据流。**在应用程序层面上、在业务层面上，我们自行定义的“数据块”在 TCP 层面上并不被协议认可。

- 半包/粘包是一个应用层问题。要解决半包/粘包问题，就是在应用程序层面建立协商一致的信息还原依据。常见的有两种方式：一是消息定长，即保证每一个完整的信息描述的长度都是一定的，这样无论 TCP/IP 协议如何进行分片，数据接收方都可以按照固定长度进行消息的还原；二是在完整的一块数据结束后增加协商一致的分隔符（例如增加一个回车符，再例如我们之前示例中一直使用的“over”关键字。

在 Java NIO 技术框架中，半包和粘包问题需要自己解决，如果使用 Netty 框架，它其中提供了多种解码器的封装帮助我们解决半包和粘包问题。甚至针对不同的数据格式，Netty 都提供了半包和粘包问题的现成解决方式，例如之前提到的 `ProtobufVarint32FrameDecoder` 解码器，就是专门解决 Protobuf 数据格式在 TCP 长连接传输时的半包问题的。

下面将会介绍 `FixedLengthFrameDecoder`、`DelimiterBasedFrameDecoder`、`LineBasedFrameDecoder` 解决半包/粘包问题的方法。由于上文中我们已经通过完整的代码演示了 Netty 的基本使用，所以下面的示例代码中为了节约篇幅，只会列出重要的代码片段。

### 1. 使用 `FixedLengthFrameDecoder` 解决问题

`FixedLengthFrameDecoder` 解码器将 TCP/IP 的数据按照指定的长度进行重新拆分，如果接收到的数据不满足设置的固定长度，Netty 将等待新的数据到达：

```
.....
serverBootstrap.childHandler(new ChannelInitializer<NioSocketChannel>() {
    @Override
    protected void initChannel(NioSocketChannel ch) throws Exception {
        ch.pipeline().addLast(new ByteArrayEncoder());
        ch.pipeline().addLast(new FixedLengthFrameDecoder(20));
        ch.pipeline().addLast(new TCPServerHandler());
        ch.pipeline().addLast(new ByteArrayDecoder());
    }
});
.....
```

Netty 上层的 `channelRead` 事件方法将在 Channel 接收到 20 个字符的情况下被触发，但是如果剩余的内容不到 20 个字符，`channelRead` 方法将不会被触发（注意 `channelReadComplete` 方法还是会被触发的）。

### 2. 使用 `LineBasedFrameDecoder` 解决问题

`LineBasedFrameDecoder` 解码器，基于最简单的“换行符”进行接收到的信息的再组织。虽然 Windows 操作系统和 Linux 操作系统中的“换行符”是不一样的，但 `LineBasedFrameDecoder` 解码器都支持。当然这个解码器没有我们后面介绍的 `DelimiterBasedFrameDecoder` 解



码器灵活。

```
.....
serverBootstrap.childHandler(new ChannelInitializer<NioSocketChannel>()
{
    @Override
    protected void initChannel(NioSocketChannel ch) throws Exception {
        ch.pipeline().addLast(new ByteArrayEncoder());
        ch.pipeline().addLast(new LineBasedFrameDecoder(100));
        ch.pipeline().addLast(new TCPServerHandler());
        ch.pipeline().addLast(new ByteArrayDecoder());
    }
});
.....
```

如果客户端发送的数据是：

```
this is 0 client \r\n request 1 \r\n"
```

那么接收方通过“换行符”重新组织后，将分两次接收到数据：

```
this is 0 client
request 1
```

### 3. 使用 DelimiterBasedFrameDecoder 解决问题

DelimiterBasedFrameDecoder 是按照“自定义”分隔符（也可以是“回车符”或者“空字符”，注意 Windows 操作系统和 Linux 操作系统中“回车符”的表示是不一样的）进行信息的重新拆分。

```
.....
serverBootstrap.childHandler(new ChannelInitializer<NioSocketChannel>()
{
    @Override
    protected void initChannel(NioSocketChannel ch) throws Exception {
        ch.pipeline().addLast(new ByteArrayEncoder());
        ch.pipeline().addLast(new DelimiterBasedFrameDecoder(1500, false,
DelimiterBasedFrameDecoder.lineDelimiter()));
        ch.pipeline().addLast(new TCPServerHandler());
        ch.pipeline().addLast(new ByteArrayDecoder());
    }
});
.....
```

DelimiterBasedFrameDecoder 有三个参数，这里介绍一下。

- maxFrameLength: 最大分割长度，如果接收方在一段长度大于 maxFrameLength 的数据

段中，没有找到指定的分隔符，那么这个处理器会抛出 `TooLongFrameException` 异常。

- `stripDelimiter`: 这是一个布尔型参数，它表示是否保留指定的分隔符。
- `delimiters`: 设置的分隔符。一般使用 `Delimiters.lineDelimiter()` 或者 `Delimiters.nullDelimiter()`。当然也可以自定义分隔符，定义成 `Bytebuf` 的类型就行了。

## 7.8 不得不提的线程池

在本章讲解 I/O 通信模型的过程中，曾多次提到线程池在 I/O 通信模型中的辅助使用。本节我们将专门讨论线程池在 Java 语言中的使用和工作方式问题。笔者在整理本书内容时，原本设想过去不去讲解一些最基础的知识点，但是在这个过程中一部分读者在笔者博客的留言，使笔者明显感觉到大家对这一部分基础知识的理解不够深入，影响到读者对本书使用线程池思路的理解，所以笔者决定还是花一定的篇幅进行介绍。如果你已经熟悉相关的知识内容，可以直接略过本节内容。

### 7.8.1 为什么要使用线程池

线程是一个操作系统级别的概念，也就是说操作系统负责线程的创建、挂起、运行、阻塞和终止操作。而操作系统创建线程、切换线程状态、终结线程都要进行 CPU 调度——这是一个耗费时间和系统资源的事情。

另一方面，目前在大多数生产环境上所面临问题的技术背景一般是：处理某一次请求的时间是非常短暂的，但是请求数量却非常巨大。在这种技术背景下，如果我们为每一个请求都单独创建一个线程，那么物理机（或者虚拟机）上的所有资源基本上都被操作系统创建线程、切换线程状态、销毁线程这些操作所占用，用于业务请求处理的资源反而减少了。所以最理想的处理方式是，将处理请求的线程数量控制在一个范围，既保证后续的请求不会等待太长时间，又保证物理机（或者虚拟机）将足够的资源用于请求处理本身。最后请注意，Linux 操作系统是有最大线程数量限制的。当运行的线程数量逼近这个值的时候，操作系统会变得不稳定。这也是我们要限制线程数量的原因（图 7-33）。

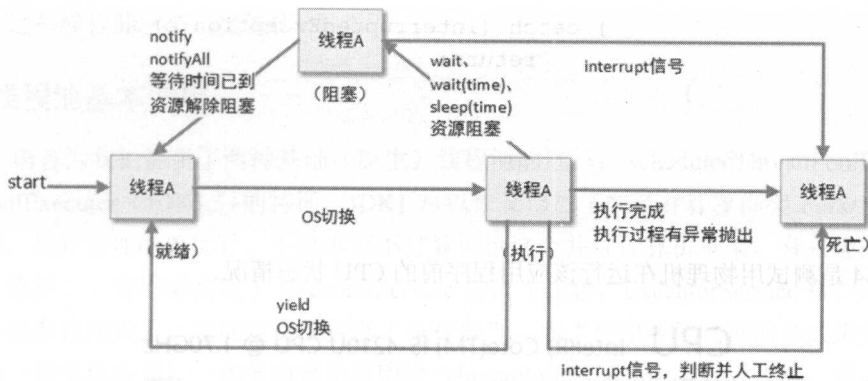


图 7-33 线程状态与切换

那么线程的切换工作到底会消耗多少 CPU 资源呢？这里我们做一个实验，验证一下线程切换中的 CPU 资源消耗情况。这个验证程序笔者给起了一个名字，叫“玩死 CPU”：我们将在一个应用程序中创建 1000 个线程，然后让这 1000 个线程不停地在“阻塞”和“就绪”两个状态间切换，以便 CPU 不停地对这些线程进行调度。代码主要片段如下所示：

```

.....
// 玩死 CPU
public class PlayDeadCPU implements Runnable {
    private int waitTime;
    public PlayDeadCPU(int waitTime) {
        this.waitTime = waitTime;
    }
    public static void main(String[] args) throws Throwable {
        // 创建 1000 个线程，这 1000 个线程的阻塞时间分别在 1~5 毫秒间
        for (int i = 1; i <= 1000; i++) {
            new Thread(new PlayDeadCPU((i % 5) + 1)).start();
        }
        // 这里的同步锁只是保证主线程不退出，和测试本身没有关系
        synchronized (PlayDeadCPU.class) {
            PlayDeadCPU.class.wait();
        }
    }
    @Override
    public void run() {
        Thread thread = Thread.currentThread();
        // 每个线程都不停地在“阻塞”和“就绪”状态间进行状态切换
        synchronized (this) {
            while(!thread.isInterrupted()) {
                try {
                    this.wait(waitTime);
                }
            }
        }
    }
}

```

```
        } catch (InterruptedException e) {  
            return;  
        }  
    }  
}
```

图 7-34 是测试用物理机在运行该应用程序前的 CPU 状态情况。

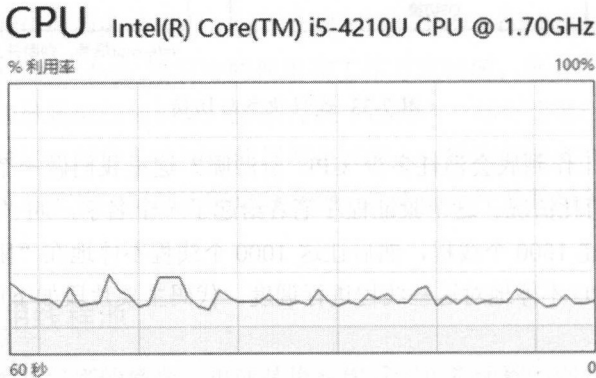


图 7-34 运行“玩死 CPU”前的 CPU 状态情况

这是一个 i5 的 CPU，之所以在运行以上程序前 CPU 还有一些使用率，是因为操作系统还有一些其他程序在运行。接着我们来运行以上代码，如图 7-35 所示。

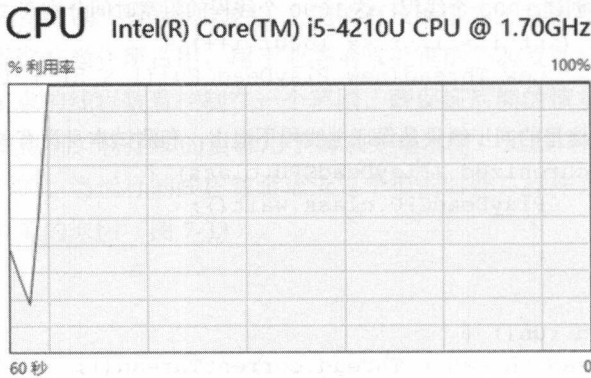


图 7-35 CPU 已死

可以看到 CPU 同时对 1000 个线程进行调度（阻塞、就绪、执行），足以消耗掉一个普通 i5 型号的 CPU 所有的空闲资源。各位读者可以看到这还不包括任何业务逻辑的处理代码，就



只进行了这些线程的状态调度操作。

## 7.8.2 线程池基本使用

Java 语言为我们提供了两种基础（原生）线程池的选择：**ScheduledThreadPoolExecutor** 和 **ThreadPoolExecutor**（JDK1.5+的特征，JDK1.7+以来又增加了支持并计算框架 Fork/Join 的 **ForkJoin Pool**，其并发性能更优异。不过本书不打算讨论这个并行计算机框架，有兴趣的读者可以查阅官方资料）。它们都实现了 **ExecutorService** 接口（注意，**ExecutorService** 接口本身和“线程池”并没有直接关系，它的定义更接近“执行器”，而“使用线程管理的方式进行实现”只是其中的一种实现方式）。本节内容主要围绕 **ThreadPoolExecutor** 类进行讲解。首先我们来看看 **ThreadPoolExecutor** 类的最简单使用方式：

```
package test.thread.pool;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
.....
public class PoolThreadSimple {
    public static void main(String[] args) throws Throwable {
        /*
         * corePoolSize: 核心大小，线程池初始化的时候，就会有这么大
         * maximumPoolSize: 线程池最大线程数
         * keepAliveTime: 如果当前线程池中线程数大于 corePoolSize
         * 多余的线程，在等待 keepAliveTime 时间后如果还没有指派新的线程任务给它，
         * 它就会被回收
         * unit: 等待时间 keepAliveTime 的单位
         * workQueue: 等待队列。这个对象的设置是本书将重点介绍的内容
         */
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(5, 10,
1, TimeUnit.MINUTES, new SynchronousQueue<Runnable>());
        for(int index = 0 ; index < 10 ; index ++){
            poolExecutor.submit(new
PoolThreadSimple.TestRunnable(index));
        }
        // 没有特殊含义，只是为了保证 Main 线程不会退出
        synchronized (poolExecutor) {
            poolExecutor.wait();
        }
    }
    // 这个就是测试用的线程
    private static class TestRunnable implements Runnable {
        private static Log LOGGER = LogFactory.getLog(TestRunnable.class);
        maximumPoolSize
```



```

// 记录任务的唯一编号, 这样在日志中好做识别
private Integer index;
public TestRunnable(int index) {
    this.index = index;
}
public Integer getIndex() {
    return index;
}
@Override
public void run() {
    // 线程中, 就只做一件事情: 等待 60 秒的时间, 以便模拟业务操作过程
    Thread currentThread = Thread.currentThread();
    TestRunnable.LOGGER.info("线程: " + currentThread.getId() + "
中的任务 (" + this.getIndex() + ") 开始执行===");
    synchronized (currentThread) {
        try {
            currentThread.wait(60000);
        } catch (InterruptedException e) {
            TestRunnable.LOGGER.error(e.getMessage(), e);
        }
    }
    TestRunnable.LOGGER.info("线程: " + currentThread.getId() + "
中的任务 (" + this.getIndex() + ") 执行完成");
}
}
}

```

### 7.8.3 ThreadPoolExecutor 逻辑结构和工作方式

在上面的代码中, 我们创建线程池的时候使用了 `ThreadPoolExecutor` 中最简单的一个构造函数:

```

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue)

```

构造函数中需要传入的参数包括 `corePoolSize`、`maximumPoolSize`、`keepAliveTime`、`timeUnit` 和 `workQueue`。要明确理解这些参数 (和后续将要介绍的参数) 的含义, 就首先要搞清楚 `ThreadPoolExecutor` 线程池的逻辑结构, 如图 7-36 所示。

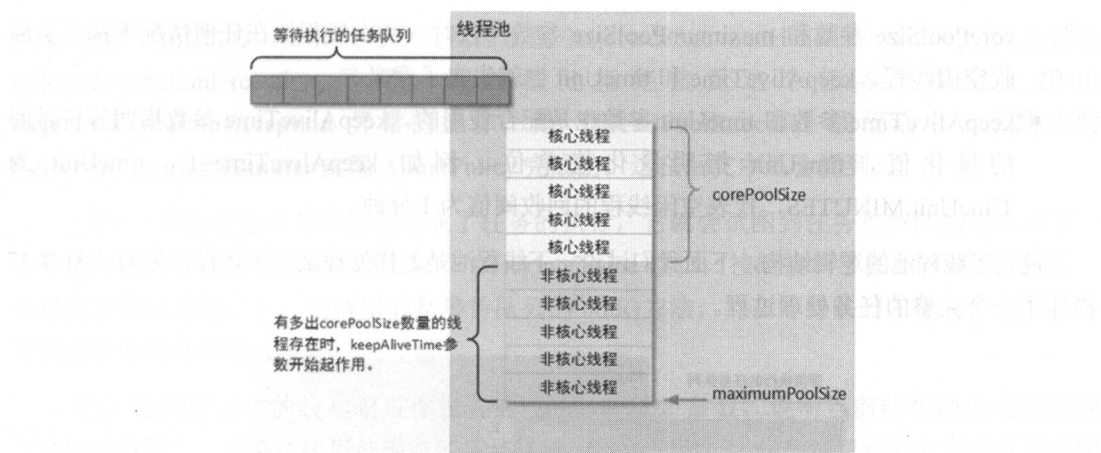


图 7-36 线程池的逻辑结构

一定要注意一个概念：存在于线程池中的容器一定是 **Thread** 对象而不是你要求运行的任务，所以这叫线程池而不叫任务池或者对象池。你要求运行的任务将被分配给这个线程池中某个空闲的线程去运行或者直接创建一个新的线程去运行。从图 7-36 中，我们可以看到构成线程池的几个重要元素。

- 等待队列：顾名思义，就是调用线程池对象的 `submit()` 方法或者 `execute()` 方法，要求线程池运行的任务（这些任务必须实现 `Runnable` 接口或者 `Callable` 接口），但是出于某些原因线程池并没有马上运行这些任务，而是送入一个队列等待执行（这些原因后文会进行讲解）。
- 核心线程：线程池主要用于执行任务的是“核心线程”，“核心线程”的数量是由创建线程时所设置的 `corePoolSize` 参数决定的。如果不进行特别的设定，线程池中始终保持 `corePoolSize` 数量的线程数（不包括创建阶段）。
- 非核心线程：一旦在等待队列中堆积的任务数量过多（由等待队列的特性决定），线程池将创建“非核心线程”临时帮助运行任务。你设置的大于 `corePoolSize` 参数小于 `maximumPoolSize` 参数的部分，就是线程池可以临时创建的“非核心线程”的最大数量。如果你的线程池中有大于 `corePoolSize` 参数所设定的线程数量，且某个线程又没有运行任何任务，那么线程池将在这个空闲线程并经过 `keepAliveTime` 时间后，将这个空闲线程进行销毁，直到线程池的线程数量重新达到 `corePoolSize`。
- 也就是说，并不是所谓的“非核心线程”才会被回收，而是谁的空闲时间达到 `keepAliveTime` 这个阈值，且没有新的任务需要执行就会被回收并销毁，直到线程池中线程数量等于 `corePoolSize` 为止。
- `maximumPoolSize` 参数也是当前线程池允许创建的最大线程数量。如果设置的

`corePoolSize` 参数和 `maximumPoolSize` 参数一致时，那么线程池在任何情况下都不会回收空闲线程。`keepAliveTime` 和 `timeUnit` 也就失去了意义。

- `keepAliveTime` 参数和 `timeUnit` 参数也是配合使用的。`keepAliveTime` 参数指明等待时间的量化值，`timeUnit` 指明量化值单位。例如 `keepAliveTime=1`，`timeUnit` 为 `TimeUnit.MINUTES`，代表空闲线程的回收阈值为 1 分钟。

说完了线程池的逻辑结构，下面我们讨论一下线程池是怎样处理某一个运行任务的。图 7-37 描述了一个完整的任务处理过程。

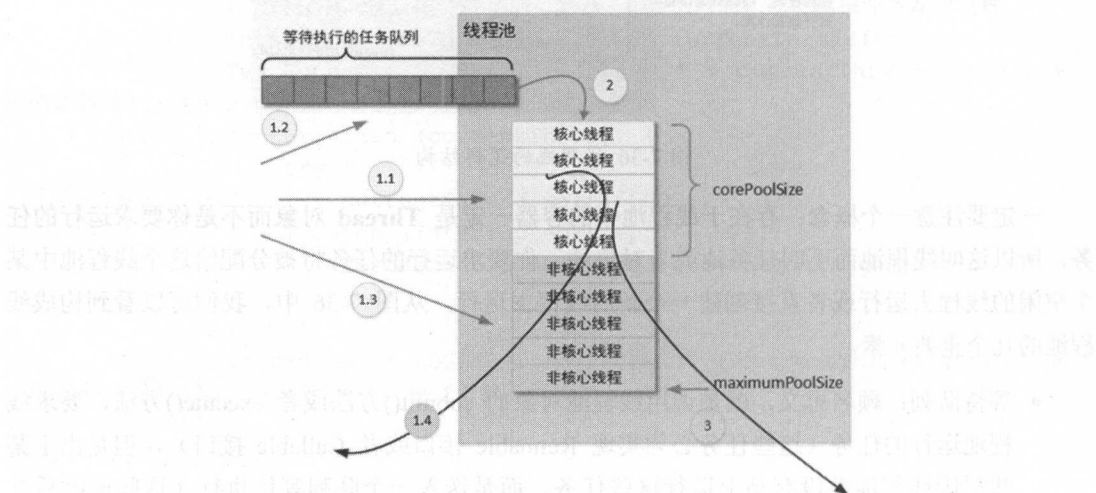


图 7-37 线程池完整的任务处理过程

(1) 首先可以通过线程池提供的 `submit()` 方法或者 `execute()` 方法，要求线程池执行某个任务。线程池收到这个要求执行的任务后，会有几种处理情况。

① 如果当前线程池中运行的线程数量还没有达到 `corePoolSize` 大小时，那么线程池会创建一个新的线程运行任务，无论之前已经创建的线程是否处于空闲状态。

② 如果当前线程池中运行的线程数量已经达到设置的 `corePoolSize` 大小，那么线程池会把这个任务加入到等待队列中，直到某一个线程空闲了，线程池会根据你设置的等待队列所具有的规则，从队列中取出一个新的任务执行。

③ 如果根据队列规则，这个任务无法加入等待队列，那么这时线程池就会创建一个“非核心线程”直接运行这个任务。注意，如果这种情况下任务执行成功，那么当前线程池中的线程数量一定大于 `corePoolSize`。

④ 如果这个任务无法被“核心线程”直接执行，又无法加入等待队列，又无法创建“非

核心线程”直接执行，且你没有为线程池设置 `RejectedExecutionHandler`，那么线程池会抛出 `RejectedExecutionException` 异常，即线程池拒绝接受这个任务（实际上抛出 `RejectedExecutionException` 异常的操作，是 `ThreadPoolExecutor` 线程池中一个默认的 `RejectedExecutionHandler` 实现 `AbortPolicy` 完成的，这在后文中还会提到）。

（2）一旦线程池中某个线程完成了任务的执行，它就会试图到任务等待队列中拿取下一个等待任务（所有的等待队列都实现了 `BlockingQueue` 接口，按照接口字面上的理解，这是一个可阻塞的队列接口），它调用的是等待队列的 `poll()` 方法，并在等待队列中没有任何新任务可以出队的情况下阻塞那里。

（3）当线程池中的线程超过你设置的 `corePoolSize` 参数，说明当前线程池中有所谓的“非核心线程”。当某个线程处理完任务且等待 `keepAliveTime` 时间后仍然没有新的任务分配给它，那么这个线程将会被回收。线程池回收线程时，对所谓的“核心线程”和“非核心线程”是一视同仁的，直到线程池中线程的数量等于读者设置的 `corePoolSize` 参数时，回收过程才会停止。

## 7.8.4 线程池的等待队列

在使用 `ThreadPoolExecutor` 线程池的时候，需要指定一个实现了 `BlockingQueue` 接口的任务等待队列。在 `ThreadPoolExecutor` 线程池的 API 文档中，一共推荐了三种等待队列，它们是 `SynchronousQueue`、`LinkedBlockingQueue` 和 `ArrayBlockingQueue`；但通过观察 `BlockingQueue` 接口的实现情况，各位读者可以发现能够直接使用的等待队列还有 `PriorityBlockingQueue`、`LinkedBlockingDeque` 和 `LinkedTransferQueue`（图 7-38）。

```

v ① BlockingQueue<E> - java.util.concurrent
    G ArrayBlockingQueue<E> - java.util.concurrent
    G BlockingArrayQueue<E> - org.eclipse.jetty.util
    GS DelayedWorkQueue - java.util.concurrent.ScheduledThreadPoolExecutor
    G DelayQueue<E extends Delayed> - java.util.concurrent
    G LinkedBlockingQueue<E> - java.util.concurrent
    > G LinkedTransferQueue<E> - org.jboss.netty.util.internal
    > G LinkedTransferQueue<E> - org.jboss.netty.util.internal
    G PriorityBlockingQueue<E> - java.util.concurrent
    G SynchronousQueue<E> - java.util.concurrent
    G VariableLinkedBlockingQueue<E> - com.rabbitmq.client.impl
    v ① BlockingDeque<E> - java.util.concurrent
        G LinkedBlockingDeque<E> - java.util.concurrent
        > ① RedisList<E> - org.springframework.data.redis.support.collections
    > ① TransferQueue<E> - scala.concurrent.forkjoin
    v ① TransferQueue<E> - java.util.concurrent
        G LinkedTransferQueue<E> - java.util.concurrent
  
```

图 7-38 `BlockingQueue` 接口实现结构

- 队列：按照大学课本《数据结构》课程中的解释，队列是一种特殊的线性结构，允许在线性结构的前端进行删除/读取操作，允许在线性结构的后端进行插入操作，这种线性结构具有“先进先出”的操作特点，如图 7-39 所示。

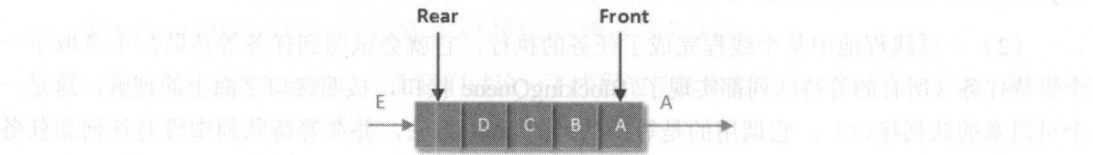


图 7-39 队列结构

但是在实际应用中，队列中的元素有可能不是以“进入的顺序”为排序依据的。例如我们要讲到的 `PriorityBlockingQueue` 队列。

- 栈：栈也是一种线性结构，但是栈和队列相比只允许在线性结构的一端进行操作，入栈和出栈都是在一端完成的（图 7-40）。

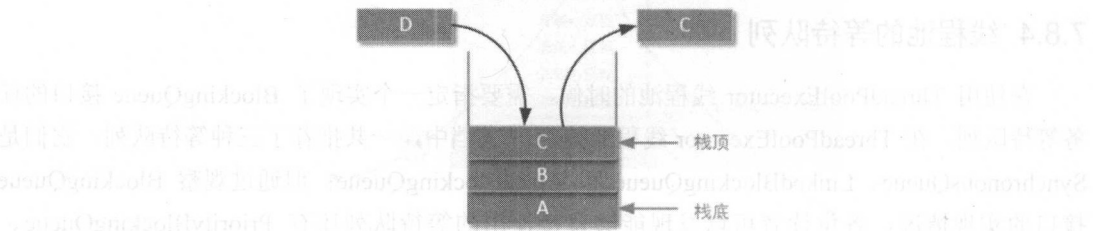


图 7-40 栈结构

**Java 线程池**主要使用队列结构存储还没有执行的`处理任务`。其中又分为有限队列和无限队列两种，所谓有限队列是指其存储的待处理任务有一个最大数量限制，当超过这个数量限制后队列将拒绝接收新任务；无限队列是指可以没有这样的数量限制（或者说大到可以忽略），只要应用程序可使用的内存足够大就可以无限制地接收待处理任务。

1. 有限队列：`SynchronousQueue`

这是一个内部没有任何容量的阻塞队列，任何一次插入操作的元素都要等待相对的删除/读取操作，否则进行插入操作的线程就要一直等待，反之亦然：

```
.....
SynchronousQueue<Object> queue = new SynchronousQueue<Object>();
// 不要使用 add 方法，因为这个队列内部没有任何容量，所以会抛出异常
// “IllegalStateException”
// queue.add(new Object());
```



```
// 操作线程会在这里被阻塞，直到有其他操作线程取走这个对象
queue.put(new Object());
.....
```

## 2. 有限队列：ArrayBlockingQueue

一个由数组支持的有限阻塞队列。此队列按 FIFO（先进先出）原则对元素进行排序。新元素插入到队列的尾部，队列获取操作则是从队列头部开始的。这是一个典型的“有界缓存区”，固定大小的数组在其中保持生产者插入的元素和使用者提取的元素。一旦创建了这样的缓存区，就不能再增加其容量。试图向已满队列中放入元素会导致操作受阻塞，试图从空队列中提取元素将导致类似阻塞。

```
.....
// 我们创建了一个 ArrayBlockingQueue，并且设置队列空间为 2
ArrayBlockingQueue<Object> arrayQueue = new ArrayBlockingQueue<Object>
(2);
// 插入第一个对象
arrayQueue.put(new Object());
// 插入第二个对象
arrayQueue.put(new Object());
// 插入第三个对象时，这个操作线程就会被阻塞
arrayQueue.put(new Object());
// 请不要使用 add 方法，它和 SynchronousQueue 的 add 操作一样使用了 AbstractQueue
// 中的 add 实现
.....
```

## 3. 无限队列：LinkedBlockingQueue

LinkedBlockingQueue 是我们在 ThreadPoolExecutor 线程池中常应用的等待队列。它可以指定容量也可以不指定容量。由于它具有“无限容量”的特性，所以本书将它归入了无限队列的范畴。实际上任何无限容量的队列/栈都是有默认容量的，这个容量就是 Integer.MAX\_VALUE。LinkedBlockingQueue 的实现是基于链表结构的，而不是类似 ArrayBlockingQueue 那样的数组。但实际使用过程中，开发人员不需要关心它的内部实现，如果你指定了 LinkedBlockingQueue 的容量大小，那么它反映出来的使用特性就和 ArrayBlockingQueue 类似（但内容实现是不一样的）。

```
LinkedBlockingQueue<Object> linkedQueue = new LinkedBlockingQueue<Object>
(2);
linkedQueue.put(new Object());
// 插入第二个对象
linkedQueue.put(new Object());
// 插入第三个对象时，这个操作线程就会被阻塞
linkedQueue.put(new Object());
```

```
// =====
// 或者如下使用:
LinkedBlockingQueue<Object> linkedQueue = new LinkedBlockingQueue
<Object>();
linkedQueue.put(new Object());
// 插入第二个对象
linkedQueue.put(new Object());
// 插入N个对象, 都不会被阻塞
linkedQueue.put(new Object());
```

#### 4. 无限队列: LinkedBlockingDeque

**LinkedBlockingDeque** 是一个基于链表的双端队列。**LinkedBlockingQueue** 的内部结构决定了它只能从队列尾部插入元素, 从队列头部取出元素; 但是 **LinkedBlockingDeque** 既可以从尾部插入/取出元素, 又可以从头部插入/取出元素。

```
.....
LinkedBlockingDeque<TempObject> linkedDeque = new LinkedBlockingDeque
<TempObject>();
// push, 可以从队列的头部插入元素
linkedDeque.push(new TempObject(1));
linkedDeque.push(new TempObject(2));
linkedDeque.push(new TempObject(3));
// poll, 可以从队列的头部取出元素
TempObject tempObject = linkedDeque.poll();
// 这里会打印 tempObject.index = 3
System.out.println("tempObject.index = " + tempObject.getIndex());

// put, 可以从队列的尾部插入元素
linkedDeque.put(new TempObject(4));
linkedDeque.put(new TempObject(5));
// pollLast, 可以从队列尾部取出元素
tempObject = linkedDeque.pollLast();
// 这里会打印 tempObject.index = 5
System.out.println("tempObject.index = " + tempObject.getIndex());
.....
```

#### 5. 无限队列: PriorityBlockingQueue

**PriorityBlockingQueue** 是一个按照优先级进行内部元素排序的无限队列。存放在 **PriorityBlockingQueue** 中的元素必须实现 **Comparable** 接口, 这样才能通过实现 **compareTo()** 方法进行排序。优先级最高的元素将始终排在队列的头部; **PriorityBlockingQueue** 不会保证优先级一样的元素的排序, 也不会保证当前队列中除了优先级最高的元素以外的元素随时处于正确的排序位置。

这是什么意思呢？换句话说来描述：**PriorityBlockingQueue** 并不保证除了队列头部以外的元素排序一定是正确的。请看下面的示例代码：

```
.....
PriorityBlockingQueue<TempObject> priorityQueue = new PriorityBlocking
Queue<TempObject>();
priorityQueue.put(new TempObject(-5));
priorityQueue.put(new TempObject(5));
priorityQueue.put(new TempObject(-1));
priorityQueue.put(new TempObject(1));

// 第一个元素是 5
// 实际上在还没有执行 priorityQueue.poll() 语句的时候，队列中的第二个元素不一定是 1
TempObject targetTempObject = priorityQueue.poll();
System.out.println("tempObject.index = " + targetTempObject.getIndex());
// 第二个元素是 1
targetTempObject = priorityQueue.poll();
System.out.println("tempObject.index = " + targetTempObject.getIndex());
// 第三个元素是-1
targetTempObject = priorityQueue.poll();
System.out.println("tempObject.index = " + targetTempObject.getIndex());
// 第四个元素是-5
targetTempObject = priorityQueue.poll();
System.out.println("tempObject.index = " + targetTempObject.getIndex());
// =====
// 这个元素类，必须实现 Comparable 接口
private static class TempObject implements Comparable<TempObject> {
    private int index;
    public TempObject(int index) {
        this.index = index;
    }
    public int getIndex() {
        return index;
    }
    @Override
    public int compareTo(TempObject o) {
        return o.getIndex() - this.index;
    }
}
.....
```

## 6. 无限队列：LinkedTransferQueue

**LinkedTransferQueue** 也是一个无限队列，它除了具有一般队列的先进先出操作特性，还具



有一个阻塞特性：LinkedTransferQueue 可以由一对生产者/消费者线程进行操作，当生产者将一个新的元素插入队列后，生产者线程将会一直等待，直到某一个消费者线程将这个元素取走，反之亦然。

LinkedTransferQueue 的操作特性可以由下面这段代码体现。在下面的代码片段中，有两种类型的线程：生产者和消费者，这两类线程互相等待对方的操作。

```
// 生产者线程
private static class ProducerRunnable implements Runnable {
    private LinkedTransferQueue<TempObject> linkedQueue;
    public ProducerRunnable(LinkedTransferQueue<TempObject> linkedQueue)
    {
        this.linkedQueue = linkedQueue;
    }
    @Override
    public void run() {
        for(int index = 1 ; ; index++) {
            try {
                // 向 LinkedTransferQueue 队列插入一个新的元素
                // 然后生产者线程就会等待，直到有一个消费者将这个元素从队列中取走
                this.linkedQueue.transfer(new TempObject(index));
            } catch (InterruptedException e) {
                e.printStackTrace(System.out);
            }
        }
    }
}

// 消费者线程
private static class ConsumerRunnable implements Runnable {
    private LinkedTransferQueue<TempObject> linkedQueue;
    public ConsumerRunnable(LinkedTransferQueue<TempObject> linkedQueue)
    {
        this.linkedQueue = linkedQueue;
    }
    @Override
    public void run() {
        Thread.currentThread = Thread.currentThread();
        while(!currentThread.isInterrupted()) {
            try {
                // 等待，直到从 LinkedTransferQueue 队列中得到一个元素
                TempObject targetObject = this.linkedQueue.take();
                System.out.println("线程 (" + currentThread.getId() + ")
取得 targetObject.index = " + targetObject.getIndex());
```

```

        } catch (InterruptedException e) {
            e.printStackTrace(System.out);
        }
    }
}

.....
// =====以下是启动代码:
LinkedTransferQueue<TempObject> linkedQueue = new LinkedTransferQueue
<TempObject>();
// 这是一个生产者线程
Thread producerThread = new Thread(new ProducerRunnable(linkedQueue));
// 这里有两个消费者线程
Thread consumerRunnable1 = new Thread(new ConsumerRunnable(linkedQueue));
Thread consumerRunnable2 = new Thread(new ConsumerRunnable(linkedQueue));
// 开始运行
producerThread.start();
consumerRunnable1.start();
consumerRunnable2.start();
// 这里只是为了主线程不退出
Thread currentThread = Thread.currentThread();
synchronized (currentThread) {
    currentThread.wait();
}
.....

```

## 7.8.5 拒绝任务

在 `ThreadPoolExecutor` 线程池中还有一个重要的接口 `RejectedExecutionHandler`。当提交给线程池的某一个新任务无法直接被线程池中的“核心线程”直接处理，又无法加入等待队列，也无法创建新的线程执行时；又或者线程池已经调用 `shutdown()` 方法停止了工作；又或者线程池不是处于正常的工作状态；这时候 `ThreadPoolExecutor` 线程池会拒绝处理这个任务，同时触发创建 `ThreadPoolExecutor` 线程池时定义的 `RejectedExecutionHandler` 接口的实现：

New tasks submitted in method execute will be rejected when the Executor has been shut down, and also when the Executor uses finite bounds for both maximum threads and work queue capacity, and is saturated. In either case, the execute method invokes the `RejectedExecutionHandler.rejectedExecution` method of its `RejectedExecutionHandler`. Four predefined handler policies are provided.

各位读者在创建 `ThreadPoolExecutor` 线程池时，一定会指定 `RejectedExecutionHandler` 接口



的实现。如果你调用的是不需要指定 `RejectedExecutionHandler` 接口的构造函数，如：

```
public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue)

public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory)
```

那么 `ThreadPoolExecutor` 线程池在创建时，会使用一个默认的 `RejectedExecutionHandler` 接口实现，源代码片段如下：

```
public class ThreadPoolExecutor extends AbstractExecutorService {
    .....
    // The default rejected execution handler
    private static final RejectedExecutionHandler defaultHandler = new
    AbortPolicy();
    .....
    // 可以看到，ThreadPoolExecutor 中的两个没有指定 RejectedExecutionHandler
    // 接口的构造函数，
    // 都使用了一个 RejectedExecutionHandler 接口的默认实现 AbortPolicy
    public ThreadPoolExecutor(int corePoolSize,
        int maximumPoolSize, long keepAliveTime,
        TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        Executors.defaultThreadFactory(), defaultHandler);
    }
    .....
    public ThreadPoolExecutor(int corePoolSize,
        int maximumPoolSize, long keepAliveTime,
        TimeUnit unit, BlockingQueue<Runnable> workQueue, Thread
    Factory threadFactory) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        threadFactory, defaultHandler);
    }
    .....
}
```

实际上，在 `ThreadPoolExecutor` 中已经提供了四种可以直接使用的 `RejectedExecutionHandler` 接口的实现。

### 1. CallerRunsPolicy

这个拒绝处理器，将直接运行这个任务的 `run` 方法。但是，并不在 `ThreadPoolExecutor` 线

程池中的线程中运行，而是直接调用这个任务实现的 `run` 方法。源代码如下：

```
.....
public static class CallerRunsPolicy implements RejectedExecutionHandler
{
    public CallerRunsPolicy() { }
    /**
     * Executes task r in the caller's thread, unless the executor
     * has been shut down, in which case the task is discarded.
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
}
.....
```

## 2. AbortPolicy

这个处理器，在任务被拒绝后会创建一个 `RejectedExecutionException` 异常并抛出。这个处理过程也是 `ThreadPoolExecutor` 线程池默认的 `RejectedExecutionHandler` 实现：

A handler for rejected tasks that throws a `RejectedExecutionException`.

## 3. DiscardPolicy

`DiscardPolicy` 处理器，将会默默丢弃这个被拒绝的任务，不会抛出异常，也不会通过其他方式执行这个任务的任何一个方法，更不会出现任何的日志提示：

A handler for rejected tasks that silently discards the rejected task.

## 4. DiscardOldestPolicy

这个处理器很有意思。它会检查当前 `ThreadPoolExecutor` 线程池的等待队列，并调用队列的 `poll()` 方法，将当前处于等待队列头部的等待任务强行取出，然后再试图将当前被拒绝的任务提交到线程池执行：

```
public static class DiscardOldestPolicy implements RejectedExecution
Handler {
    .....
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            e.getQueue().poll();
            e.execute(r);
        }
    }
}
```

```

    }
    }
    .....
}

```

实际上查阅这四种 `ThreadPoolExecutor` 线程池自带的拒绝处理器实现，读者可以发现 `CallerRunsPolicy`、`DiscardPolicy`、`DiscardOldestPolicy` 处理器针对被拒绝的任务都不是一个很好的处理方式。

`CallerRunsPolicy` 在非线程池以外直接调用任务的 `run` 方法，可能会造成线程安全和线程控制方面的问题；`DiscardPolicy` 默默地忽略掉被拒绝任务，也没有输出日志或者提示，开发人员不会知道线程池的处理过程出现了错误，这也是不科学的；`DiscardOldestPolicy` 中的 `e.getQueue().poll()` 方式好像是合理的，但是如果等待队列出现了容量问题，那么大多数情况下就是这个线程池的代码出现了 Bug。合适的处理方式还是 `AbortPolicy` 提供的：抛出异常，由开发人员进行处理。

## 7.8.6 ThreadPoolExecutor 中常用属性总结

本节我们利用 `ThreadPoolExecutor` 的源代码，来总结 `ThreadPoolExecutor` 线程池中已经介绍过和没有介绍过的属性。当然在实际工作中，你不需要随时记住这些属性的含义，但是理解这些属性的意义，至少可以在线程池出现状况时，找到解决问题的突破口。

```

.....
//等待队列，我们已经花了很大的篇幅来介绍线程池的等待队列
private final BlockingQueue<Runnable> workQueue;
// ReentrantLock 是 JDK 1.5+中引入的一个乐观锁
// 线程池主要在创建线程、回收线程、终止线程等操作的时候，使用 ReentrantLock，保证线
// 程池中对象的状态一致性
private final ReentrantLock mainLock = new ReentrantLock();
//线程池中的一个任务就是一个 Worker，这个集合用于存储正在线程池中运行的任务
private final HashSet<Worker> workers = new HashSet<Worker>();
//这个参数记录线程池曾经达到过的最大的“池大小”
private int largestPoolSize;
//这个属性记录线程池已经完成的任务数量
private long completedTaskCount;
/*
 * All user control parameters are declared as volatiles so that
 * ongoing actions are based on freshest values, but without need
 * for locking, since no internal invariants depend on them
 * changing synchronously with respect to other actions.
 */
//创建线程使用的线程工厂

```

```

private volatile ThreadFactory threadFactory;
//指定的“拒绝处理器”
//系统中还有一个默认的“拒绝处理器”：defaultHandler =new AbortPolicy();
private volatile RejectedExecutionHandler handler;
//空闲线程在等待工作时间超时
private volatile long keepAliveTime;
//是否在超过等待时间后，就连线程池中小于 corePoolSize 的“核心线程”对象也进行回收
//默认情况为 false
private volatile boolean allowCoreThreadTimeOut;
//“核心线程”的大小。小于或者等于这个数量的线程，即便超过 keepAliveTime 也不会被回
//收，除非 allowCoreThreadTimeOut 被设置为 true
private volatile int corePoolSize;
//线程池中最大的线程数量
private volatile int maximumPoolSize;
//默认的“拒绝处理器”
private static final RejectedExecutionHandler defaultHandler =new
AbortPolicy();

```



## 第 8 章

# RPC 与系统间调用

好的网络 I/O 模型是构成高性能系统的基础之一，而搭建在其上的各种 RPC 框架则是完成高性能系统间通信的一种具体技术形式。本章开始部分将向读者介绍狭义的 RPC 定义和广义化的 RPC 定义，接着讲述 RPC 的技术原理和一款流行的 RPC 技术——Apache Thrift 的使用方法和工作过程。

## 8.1 RPC 技术原理

### 8.1.1 什么是 RPC

RPC (Remote Procedure Call Protocol, 远程过程调用协议)，一个通俗的描述是：客户端在不知道调用细节的情况下，调用存在于远程计算机上的某个对象，就像调用本地应用程序中的对象一样。更为通俗易懂的描述是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。通过以上的描述，我们至少可以从中挖掘出几个要点。

- RPC 是协议：既然是协议就只是一套规范规则，也就需要有人遵循这套规范来进行实现。目前典型的 RPC 实现包括：dubbo（注意是小写的，不是大写的 DUBBO 服务治理框架）、Apache Thrift、GRPC、Hetty 等。这里要说明一下，从目前技术的发展趋势来看，实现了 RPC 协议的应用组件往往都会附加其他重要功能，例如 dubbo 是服务治理框架 DUBBO 中使用的一个内部 RPC 实现，DUBBO 还包括了服务发现、访问权限控制等功能。
- 网络协议和网络 I/O 模型对其透明：既然 RPC 的客户端认为自己是在调用本地对象，那么传输层使用的是 TCP 还是 HTTP 协议，又或者是一些其他的网络协议它就不需要



关心了。既然网络协议对其透明，那么在调用过程中，使用的是哪一种网络 I/O 模型，调用者也不需要关心（但实际上还是要关心的，因为要做性能调优）。

- 信息格式对其透明：我们知道在本地应用程序中，对于某个对象的调用需要传递一些参数，并且会返回一个调用结果。至于被调用的对象内部是如何使用这些参数，并计算出处理结果的，调用方是不需要关心的。那么对于远程调用来说，这些参数会以某种信息格式传递给网络上的另外一台计算机，这个信息格式是怎样构成的，调用方就不需要关心了，如图 8-1 所示。
- RPC 框架都应该有跨语言能力，为什么这样说呢？因为调用方实际上也不清楚远程服务器的应用程序是使用什么语言运行的。那么对于调用方来说，无论服务器方使用的是什么语言，本次调用都应该成功，并且返回值也应该按照调用方程序语言所能理解的形式进行描述。当然，现实情况下由于一些 RPC 框架的特殊工作场景，也没有强行要求其提供跨语言能力，例如只工作在 Java 语言下的 RMI 就是这样一套 RPC 框架。

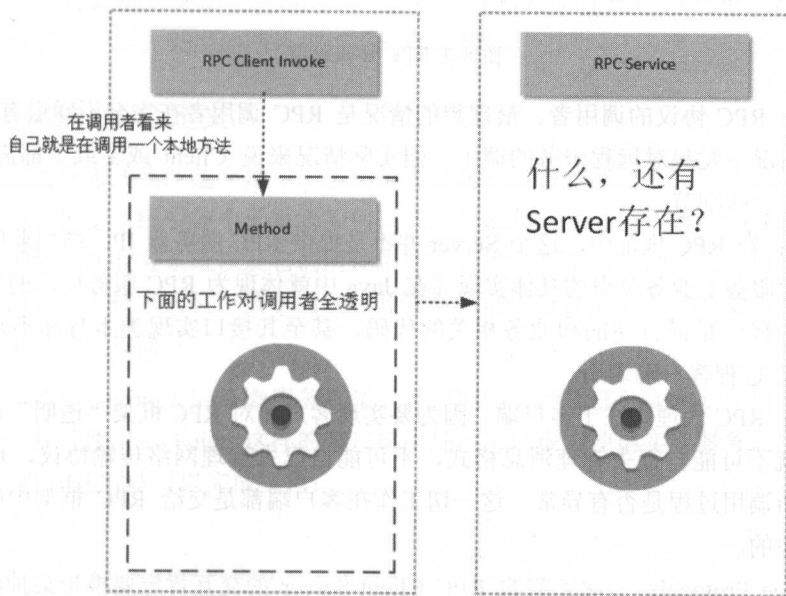


图 8-1 从调用者角度看 RPC 协议

## 8.1.2 RPC 要素

8.1.1 节描述的 RPC 特点是作为 RPC 的调用者所观察到的。实际调用环境下 RPC 的调用者还是需要知道一些调用细节的。既然我们是要讲解 RPC 的基本概念，那么 RPC 协议内部是怎样一个结构要求就要说一说了（图 8-2）。

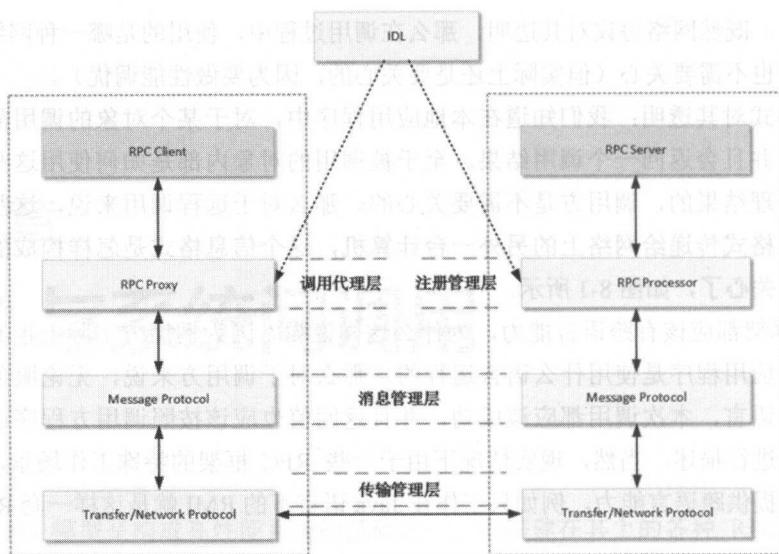


图 8-2 RPC 协议结构

- **Client:** RPC 协议的调用者。最理想的情况是 RPC 调用者在完全不知道有 RPC 框架存在的情况下发起对远程服务的调用。但实际情况来说 Client 或多或少都需要指定 RPC 框架的一些细节。
- **Server:** 在 RPC 规范中, 这个 Server 并不是提供 RPC 服务器 IP、端口监听的模块。而是远程服务上业务逻辑的具体实现 (在 Java 中就体现为 RPC 服务接口的具体实现)。其中的代码是最普通的和业务相关的代码, 甚至其接口实现类本身都不知道将被某一个 RPC 远程客户端调用。
- **Proxy:** RPC 代理存在于客户端, 因为要实现客户端对 RPC 框架“透明”调用, 那么客户端就不可能自行去管理消息格式, 不可能自己去管理网络传输协议, 也不可能自己去判断调用过程是否有异常。这一切工作在客户端都是交给 RPC 框架中的“代理”层来处理的。
- **Message Protocol:** 一次完整的 RPC Client-Server 的交互肯定要携带某种两端都能识别的, 共同约定的消息格式。RPC 的消息管理层专门对网络传输所承载的信息进行编号和解码操作。目前流行的技术趋势是不同的 RPC 实现, 为了加强自身框架的效率都有一套 (或者几套) 私有的消息格式。例如后文我们将讲解的一套 RPC 框架 Apache Thrift, 就拥有私有化的消息协议。当然 Apache Thrift 还支持通用的消息格式, 如 JSON)。
- **Transfer/Network Protocol:** 传输协议层负责管理 RPC 框架所使用的网络协议、网络 I/O 模型。例如 Hessian 的传输协议基于 HTTP (应用层协议); 再例如 Apache Thrift 的传

输协议基于 TCP（传输层协议）。传输层还需要统一 RPC 客户端和 RPC 服务端所使用的网络 I/O 模型。

- **Processor**: 存在于 RPC 服务端，由于服务器端某一个 RPC 接口的实现特性（它并不知道自己是一个将要被 RPC 提供给第三方系统调用的服务）。所以在 RPC 框架中应该有一种“负责执行 RPC 接口实现”的角色。它的职责包括管理 RPC 接口的注册、判断客户端的请求权限、控制接口实现类的执行在内的各种工作。
- **IDL**: 虽然 IDL（接口定义语言）并不是 RPC 实现中所必须的。但是需要跨语言的 RPC 框架一定会有 IDL 部分的存在。这是因为要找到一个各种语言能够理解的消息结构、接口定义的描述形式。如果你的 RPC 实现没有考虑跨语言性，那么 IDL 部分就不需要。例如 Java RMI 就是一种专门在 Java 语言间进行使用的特殊的 RPC 框架，它设计之初就没有要求要有跨语言执行特性，所以 Java RMI 没有相应的 IDL。
- 不同的 RPC 实现都有一定设计差异。例如生成 Proxy 的方式不一样、IDL 描述语言的语法不一样、服务注册的管理方式不一样、运行服务实现的方式不一样、采用的消息格式封装不一样、采用的网络协议不一样，等等。

### 8.1.3 更泛化的 RPC 定义

由于现在的分布式系统越来越多，虽然很多分布式业务系统达不到亚马逊、淘宝、京东这样的业务规模，但是系统中各个子系统如何完成通信还是一个必须要解决的问题。目前的发展趋势是，很多分布式业务系统以及它们所使用的服务治理框架都鼓励简化调用过程。按照这个发展趋势来说，很多技术人员、技术团队、开源组织都更愿意基于 HTTP 协议来完成系统间调用，并让这个过程更加简单。

最典型的就是 Spring Cloud 微服务框架，其中的服务治理部分就主要支持 HTTP 协议的系统接口注册。并且使用了 Feign 这样的客户端组件，可以很方便地实现服务调用——没有 IDL、没有 Proxy，几句注解就可以完成工作：

```
.....
@FeignClient(
    value="ServiceSystemA" ,
    fallback=ServiceSystemAClientFallback.class
)
public interface IServiceSystemA {
    //调用接口定义，由 ServiceSystemA 系统提供的 HTTP 形式的服务接口
    @RequestMapping(method=RequestMethod.POST, value= "/query
SomeList")
    public List<String> querySomeList();
}
.....
```

## 8.1.4 典型的 RPC 框架介绍

- **Java RMI:** RPC 概念最早由 SUN 提出, 后来由 IETF ONC 修订。在 Java 的早期版本中有一个典型的 RPC 协议实现叫作 RMI, 只不过 RMI 没有跨语言特性, 所以也就没有 IDL 的存在。在现在各种 RPC 框架大行其道的今天, RMI 仍然具有非常高的执行效率, 并且由于没有 IDL 的存在, 所以在构建一套完整的 RPC 实现时比其他 RPC 框架少了一些步骤, 使用起来还比较简单。如果你的业务需求中并不存在跨语言的考虑, 并且主要系统基本上都使用 Java 实现, 那么 RMI 是一个可以考虑的子系统间的调用方案。
- **GRPC:** 这是一个高性能、通用的开源 RPC 框架, 由 Google 主要面向移动应用开发并基于 HTTP/2 协议 (注意是 HTTP/2 协议, 不是我们常使用的 HTTP 1\_1 协议。HTTP/2 协议的详细介绍可以参见官方网站: <https://http2.github.io/>) 标准而设计, 基于 Protocol Buffers (Protocol Buffers) 序列化协议 (格式) 开发, 且支持众多开发语言。为了支持 GRPC 的跨语言性, GRPC 有一套独立存在的 IDL 语言。不过由于 GRPC 是 Google 的开源产品, 在信息格式封装方面 Google 主要还是推广自己的 Protocol Buffers, 所以 GRPC 是不支持其他信息格式的 (至少 Protocol Buffers 的工作性能是大家有目共睹的)。关于 GRPC 详细的使用介绍, 可以参见官方网站: <https://github.com/grpc/grpc>。
- **Thrift:** Thrift 是 Facebook 的一个开源项目, 后来进入 Apache 进行孵化。Apache Thrift 也支持跨语言特性, 所以它有自己的一套 IDL。目前它支持几乎所有主流的编程语言: C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、JavaScript、Node.js、Smalltalk、OCaml and Delphi, 等等。Apache Thrift 可以支持多种消息格式, 除了 Thrift 私有的二进制编码规则和一种 LVQ (类似于 TLV 消息格式) 的消息格式, 还有常规的 JSON 格式。Thrift 的网络协议建立在 TCP 协议基础上, 并且支持阻塞式 I/O 模型和多路复用 I/O 模型, 我们将在后文讨论 Apache Thrift 的使用。Thrift 也是目前最流行的 RPC 框架之一, 从网络上各种性能测试的情况看, Thrift 的性能都是领先的。Thrift 的官网网址为: <http://thrift.apache.org/>。
- **Netty:** Netty 是一款构建于 Netty 和 Hessian 基础上的高性能的 RPC 框架。在之前的章节中, 本书已经较详细地讲述了使用 Netty 进行网络处理和直接使用 Java 原生的 I/O 模型进行网络处理所带来的好处。Netty 的网络协议基于 HTTP, 由于采用了 Netty, 所以 Netty 支持阻塞式 I/O 模型和多路复用 I/O 模型。Netty 的消息格式采用私有的二进制流格式。
- **其他的 RPC 框架:** 除了上述的 RPC 协议的实现, 还有 Wildfly、Hprose, 等等。Hprose 是一款由国人主导的 RPC 实现, 感兴趣的读者可以去看看 (<http://www.hprose.com/>)。另外基于 RPC 的定义, Xfire、CXF 这些 Web Service 框架也可以算作 RPC 的实现 (更广义的 RPC 定义)。WSDL 描述文件可以看成是它们的 IDL, 通过 WSDL 为不同的编程语言生成 Proxy/ Processor, 通过不同的 Web 服务器管理具体服务实现的运行过程,

HTTP 是它们的通信协议，XML/JSON 是它们的统一的消息格式。

### 8.1.5 RPC 框架的性能依据

RPC 框架的性能依据如图 8-3 所示。

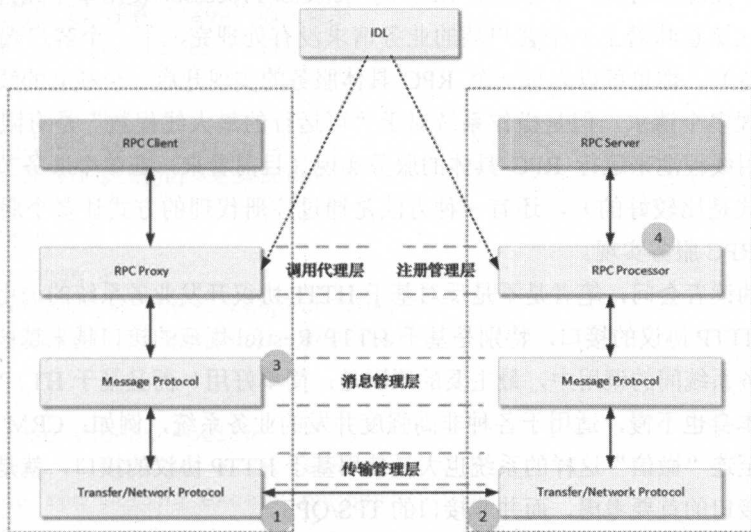


图 8-3 RPC 框架的性能依据

在物理服务器性能相同的情况下，以下几个因素会对一款 RPC 框架的性能产生直接影响。

- 所支持的网络 I/O 模型：实现了 RPC 协议的框架，如果只可以支持传统的阻塞式同步 I/O，或者只是在做一些调整后支持非阻塞式同步，那么对于实现了对多路复用 I/O 模型的 RPC 框架而言，在高并发状态下的处理性能是完全不同的。特别是单位处理性能下对内存、CPU 资源的使用率。
- 基于的网络协议：一般来说可以选择让 RPC 使用应用层协议进行工作，例如 HTTP 或者之前我们提到的 HTTP/2 协议；或者使用 TCP 协议让 RPC 框架工作在传输层。工作在哪一层网络上会对 RPC 框架的工作性能产生一定的影响，例如工作在 TCP 协议下的 RPC 框架性能普遍好于工作在 HTTP 等应用层协议下的 RPC 框架，但后者灵活性又要好得多。设置可以抛开 IDL——因为目前基本上所有主流的开发语言都可以使用 HTTP 协议。目前并没有采用 UDP 协议作为主要的传输协议的 RPC 框架。
- 选择的消息封装格式：选择或者定义一种消息格式的封装，要考虑的问题包括消息的易读性、描述单位内容时的消息体大小、编码难度、解码难度、解决半包/粘包问题的难易度。当然如果你只是想定义一种 RPC 专用的消息格式，那么消息的易读性可能不



是最需要考虑的。消息封装格式的设计是目前各种 RPC 框架性能差异的最重要原因，这就是为什么几乎所有主流的 RPC 框架都会设计私有的消息封装格式的原因。本书后续内容将举例说明在 Apache Thrift 中对消息进行二进制封装的方式。

- 实现的服务管理方式：在高并发请求下，如何管理注册的服务并且运行一次服务调用，也是一个性能影响点。你可以让 RPC 的 Selector/Processor 使用单个线程运行服务的具体实现（这意味着上一个客户端的业务请求没有处理完，下一个客户端的业务请求就需要等待），你也可以为每一个 RPC 具体服务的实现开启一个独立的线程运行（可以一次处理多个请求，但是操作系统对于“可运行的最大线程数”是有限制的），你还可以使用线程池来运行 RPC 具体的服务实现（目前看来，在单个服务节点的情况下，这种方式是比较好的），还有一种方法是通过注册代理的方式让多个服务节点来运行具体的 RPC 服务实现。
- 那么有的读者会问，笔者是不是反对基于 HTTP 协议开发业务系统的调用接口呢？当然不是，HTTP 协议的接口，特别是基于 HTTP Restful 规范的接口越来越被广泛的使用在各种业务系统间的调用中，最主要的原因是：简单好用！而且基于 HTTP 协议的接口调用速度本身也不慢，适用于各种非高强度并发的业务系统，例如：CRM 系统、财务系统，甚至连“微信”这样的系统也大量使用基于 HTTP 协议的接口，就是因为简单好用是这些接口的首要考虑，而并非接口的 TPS/QPS。

## 8.2 RPC 实践：Apache Thrift 基本使用

Thrift 最初由 Facebook 开发用作系统内各语言之间的 RPC 框架。2007 年由 Facebook 贡献到 Apache 基金，2008 年 5 月进入 Apache 孵化器，称为 Apache Thrift。和其他 RPC 实现相比，Apache Thrift 主要的优点是：支持的语言多（C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、Smalltalk 等）、并发性能高、易于编程、易于扩展。

为了支持多种语言，Apache Thrift 有一套自己的接口定义语言，并且通过 Apache Thrift 的代码生成程序，能够生成各种编程语言的代码。这样是保证各种语言进行通信的前提条件。为了能够实现简单的 Apache Thrift 实例，首先我们就需要讲解一下 Apache Thrift 的 IDL。

如果你是在 Windows 环境下运行 Apache Thrift 的，那么无须安装任何工具直接下载 Apache Thrift 在 Windows 下的代码生成程序 <http://www.apache.org/dyn/closer.cgi?path=/thrift/0.9.3/thrift-0.9.3.exe>（本章节整理时，使用的是 Apache Thrift 的 0.9.3 版本，目前应该有更新的版本了）；如果你运行在 Linux 系统下，那么下载 <http://www.apache.org/dyn/closer.cgi?path=/thrift/0.9.3/thrift-0.9.3.tar.gz>，并进行编译、安装（过程很简单，这里就不再赘述了），

安装后记得添加运行位置到环境变量中。

## 8.2.1 IDL 格式概要

以下是一个简单的 IDL 文件定义：

```
# 命名空间的定义，注意“java”的关键字
namespace java testThrift.iface
# 结构体定义
struct Request {
    1:required string paramJSON;
    2:required string serviceName;
}
# 另一个结构体定义
struct Reponse {
    1:required RESCODE responseCode;
    2:required string responseJSON;
}
# 异常描述定义
exception ServiceException {
    1:required EXCCODE exceptionCode;
    2:required string exceptionMess;
}
# 枚举定义
enum RESCODE {
    _200=200;
    _500=500;
    _400=400;
}
# 另一个枚举
enum EXCCODE {
    PARAMNOTFOUND = 2001;
    SERVICENOTFOUND = 2002;
}
# 服务定义
service HelloWorldService {
    Reponse send(1:Request request) throws (1:ServiceException e);
}
```

以上 IDL 文件是可以直接用来生成各种语言的代码的。下面给出常用的各种不同语言的代码生成命令：

```
# 生成 java
thrift-0.9.3 -gen Java ./demoHello.thrift
# 生成 C++
```

```
thrift-0.9.3 -gen cpp ./demoHello.thrift
# 生成 PHP
thrift-0.9.3 -gen php ./demoHello.thrift
# 生成 Node.js
thrift-0.9.3 -gen js:node ./demoHello.thrift
# 生成 C#
thrift-0.9.3 -gen csharp ./demoHello.thrift
# 你可以通过以下命令查看生成命令的格式
thrift-0.9.3 -help
```

## 1. 基本类型

基本类型就是不论哪一种编程语言，都支持的数据形式表现。Apache Thrift 中支持以下几种基本类型。

- bool: 布尔值（true or false），占用一个字节（one byte）
- byte: 有符号字节
- i16: 16 位有符号整型
- i32: 32 位有符号整型
- i64: 64 位有符号整型
- double: 64 位浮点型
- string: 字符串/字符数组
- binary: 二进制数据（在 Java 中表现为 java.nio.ByteBuffer）

## 2. struct 结构

在面向对象语言中，表现为“类定义”；在弱类型语言、动态语言中，表现为“结构/结构体”。定义格式如下：

```
struct <结构体名称> {
    <序号>:[字段性质] <字段类型> <字段名称> [= <默认值>] [;|,]
}
// 例如
struct Request {
    1:required binary paramJSON;
    2:required string serviceName
    3:optional i32 field1 = 0;
    4:optional i64 field2,
    5: list<map<string , string>> fields3
}
```

- 结构体名称：可以按照业务需求，给定不同的名称（区分大小写）。但是要注意，一组 IDL 定义文件中结构体名称不能重复，且不能使用 IDL 已经占用的关键字（例如

required、struct 等单词)。

- 序号：序号非常重要。它是正整数，并且按照顺序排列使用。它在 Apache Thrift 进行序列化的时候被使用。
- 字段性质：包括两种关键字 required 和 optional，如果不指定，那么系统会默认为 required。required 表示这个字段必须有值，并且 Apache Thrift 在进行序列化时，这个字段就会被序列化；optional 表示这个字段不一定有值，且 Apache Thrift 在进行序列化时，这个字段只在有值的情况下才会被序列化。
- 字段类型：在 struct 中，字段类型可以是某一个基础类型，可以是某一个之前定义好的 struct，也可以是某种 Apache Thrift 支持的容器 (set、map、list)，还可以是定义好的枚举。字段的类型是必须指定的。
- 字段名称：字段名称区分大小写，不能重复，且不能使用 IDL 已经占用的关键字 (例如 required、struct 等单词)。
- 默认值：你可以为某一个字段指定默认值 (也可以不指定)。
- 结束符：在 struct 中，支持两种结束符，可以使用 “;” 或者 “，”。当然也可以不使用结束符，Apache Thrift 代码生成程序时，会自己识别到。

### 3. containers 集合/容器

Apache Thrift 支持三种类型的容器，容器在各种编程语言中普遍存在。

list< T >: 有序列表 (Java 语言中表现为 ArrayList)，T 可以是某种基础类型，可以是某一个之前定义好的 struct，也可以是某种 Apache Thrift 支持的容器 (set、map、list)，还可以是定义好的枚举。有序列表中的元素允许重复。

set< T >: 无序元素集合 (Java 语言中表现为 HashSet)，T 可以是某种基础类型，也可以是某一个之前定义好的 struct，还可以是某种 Apache Thrift 支持的容器 (set、map、list)，还可以是定义好的枚举。无序元素集合中的元素不允许重复，一旦重复后一个元素将覆盖前一个元素。

Map< T, V >: Key—Value 的键值对应结构，在 Java 语言中表现为 HashMap。

### 4. enmu 枚举

```
enum <枚举名称> {
    <枚举字段名> = <枚举值>[;|,]
}
// 例如
enum RESCODE {
    _200=200;
    _500=500;
    _400=400;
}
```



## 5. 常量定义

Apache Thrift 允许定义常量。常量的关键字为“const”，常量的类型可以是 Apache Thrift 的基础类型，也可以是某一个之前定义好的 struct，也可以是某种 Apache Thrift 支持的容器（set、map、list），还可以是定义好的枚举。几个示例如下：

```
const i32 MY_INT_CONST = 111111;
const i64 MY_LONG_CONST = 111111222222233333333344444444;
const RESCODE MY_RESCODE = RESCODE._200;
```

## 6. exception 异常

Apache Thrift 的 exception 异常，主要在定义服务接口时使用。其定义方式类似于 struct（可以理解成把 struct 关键字换成 exception 关键字即可），示例如下：

```
exception ServiceException {
    1:required EXCCODE exceptionCode;
    2:required string exceptionMess;
}
```

## 7. service 服务接口

service 服务接口是 Apache Thrift 中最重要的 IDL 定义之一。在后续的代码生成阶段，通过 IDL 定义的这些服务接口将构成 Apache Thrift 客户端调用 Apache Thrift 服务端的基本远端过程。service 服务接口的定义形式如下所示：

```
service <服务名称> {
    <void | 返回指类型> <服务方法名>([<入参序号>:[required | optional] <参数类型> <参数名> ...]) [throws ([<异常序号>:[required | optional] <异常类型> <异常参数名>...])]
}
// 例如
service HelloWorldService {
    Reponse send(1:Request request) throws (1:ServiceException e);
}
```

- 服务名称：服务名称可以按照业务需求自行制定，注意服务名称是区分大小写的。IDL 中服务名称只有两个限制，就是不能重复使用相同的名称，不能使用 IDL 已经占用的关键字（例如 required、struct 等单词）。
- 返回值类型：如果这个调用方法没有返回类型，那么可以使用关键字“void”；返回值可以是 Apache Thrift 的基础类型，可以是某一个之前定义好的 struct，也可以是某种 Apache Thrift 支持的容器（set、map、list），还可以是定义好的枚举。
- 服务方法名：服务方法名可以根据业务需求制定，注意区分大小写。在同一个服务中，不能重复使用一个服务方法名命名多个方法（一定要注意），不能使用 IDL 已经占用



的关键字。

- 服务方法参数：<入参序号>:[required | optional] <参数类型> <参数名>。注意和 struct 中的字段定义相似，可以指定 required 或者 optional；如果不指定则系统默认为 required。如果一个服务方法中有多个参数名，那么这些参数名称不能重复。
- 服务方法异常：throws ([<异常序号>:[required | optional] <异常类型> <异常参数名>)。throws 关键字是服务方法异常定义的开始点。在 throws 关键字后面，可以定义 1 个或者多个不同的异常类型。

## 8. namespace 命名空间

Apache Thrift 支持为不同语言制定不同的命名空间：

```
namespace java testThrift.iface
namespace php testThrift.iface
namespace cpp testThrift.iface
```

## 9. 注释

Apache Thrift 支持多种风格的注释，这是为了适应不同语言背景的开发者：

```
/*
 * 注释方式 1:
 **/
// 注释方式 2
# 注释方式 3
```

## 10. include 关键字

如果整个工程中有多个 IDL 定义文件（IDL 定义文件的文件名可以随便取），那么可以使用 include 关键字，在 IDL 定义文件中，引入一个其他的 IDL 文件：

```
include "other.thrift"
```

请注意，一定使用双引号（不要使用中文全角状态下的双引号），并且不使用“;”或者“,” 结束符。

以上就是 IDL 的基本语法，由于篇幅原因不可能把每种语法、每一个细节都讲到，但是以上的语法要点已经足够你编辑一个适应业务的、灵活的 IDL 文件定义了。如果需要了解更详细的 Thrift IDL 语法，可以参考官方文档的讲述：<http://thrift.apache.org/docs/idl>。

### 8.2.2 简单的 Apache Thrift 代码

定义 Thrift 中业务接口 HelloWorldService.Iface 的实现：

```
.....
//我们定义了一个 HelloWorldService.Iface 接口的具体实现
```

```

//注意, 这个父级接口: HelloWorldService.Iface, 是由 thrift 的代码生成工具生成的
//要运行这段代码, 请导入 maven-log4j 的支持。否则修改 LOGGER.info 方法
public class HelloWorldServiceImpl implements Iface {
    //在接口定义中, 只有这个方法需要实现
    //你可以理解成这个接口的方法接受客户端的一个 Request 对象, 并且在处理完成后向客户端返
    //回一个 Reponse 对象
    //Request 对象和 Reponse 对象都是由 IDL 定义的结构,
    //并通过“代码生成工具”生成相应的 Java 代码
    @Override
    public Reponse send(Request request) throws TException {
        // 这里就是进行具体的业务处理了
        String json = request.getParamJSON();
        String serviceName = request.getServiceName();
        LOGGER.info("得到的 json: " + json + " ; 得到的 serviceName: " +
serviceName);
        // 构造返回信息
        Reponse response = new Reponse();
        response.setResponseCode(RESCODE._200);
        response.setResponseJSON("{\"user\":\"yinwenjie\"}");
        return response;
    }
}
.....

```

各位读者可以看到, 上面一段代码中的具体业务、过程与普通的业务代码没有任何区别。甚至这段代码的实现都不知道自己将被 **Apache Thrift** 框架中的客户端调用。

- 然后我们开始书写 **Apache Thrift** 的服务器端代码:

```

.....
public static final int SERVER_PORT = 9111;
public void startServer() {
    try {
        HelloBoServerDemo.LOGGER.info("看到这句就说明 thrift 服务端准备工作 ....");
        // 服务执行控制器 (主要是调度服务的具体实现该如何运行)
        TProcessor tprocessor = new HelloWorldService.Processor
<Iface>(new HelloWorldServiceImpl());
        // 基于阻塞式同步 I/O 模型的 Thrift 服务, 正式生产环境不建议用这个
        TServerSocket serverTransport = new TServerSocket (HelloBoServer
Demo. SERVER_PORT);
        // 为这个服务器设置对应的 I/O 网络模型、设置使用的消息格式封装、设置线程
        // 池参数
        Args tArgs = new Args(serverTransport);
        tArgs.processor(tprocessor);
        tArgs.protocolFactory(new TBinaryProtocol.Factory());
    }
}

```

```

        tArgs.executorService(Executors.newFixedThreadPool(100));
        // 启动这个 Thrift 服务
        TThreadPoolServer server = new TThreadPoolServer(tArgs);
        server.serve();
    } catch (Exception e) {
        HelloBoServerDemo.LOGGER.error(e);
    }
}

public static void main(String[] args) {
    HelloBoServerDemo server = new HelloBoServerDemo();
    server.startServer();
}

.....

```

**TBinaryProtocol:** 这个类代表 ache Thrift 特有的一种二进制描述格式。它的特点是传输单位数据量所使用的传输量更少。Apache Thrift 还支持多种数据格式，例如我们熟悉的 JSON 格式。本章节后续的内容中，将介绍 Apache Thrift 中的数据格式。

**tArgs.executorService():** 是不是觉得这个 `executorService` 很熟悉，是的，这个就是 Java JDK 1.5 以后 `java.util.concurrent` 包提供的异步任务调度服务接口，Java 标准线程池 `ThreadPoolExecutor` 就是它的一个实现。

**server.serve(),** 由于是使用的阻塞式同步网络 I/O 模型，所以这个应用程序的主线程执行到这句话以后就会保持阻塞状态了。只要下层网络状态不出现错误，这个线程就会一直停在这里。

- 接下来我们进行最简单的 Apache Thrift Client 的代码编写：

```

// 同样是基于同步阻塞模型的 Thrift Client
public class HelloClient {
    .....
    public static final void main(String[] args) throws Exception {
        // 服务器所在的 IP 和端口
        TSocket transport = new TSocket("127.0.0.1", 9111);
        TProtocol protocol = new TBinaryProtocol(transport);
        // 准备调用参数
        Request request = new Request("{\"param\":\"field1\"}", "\\myService\\queryService");
        HelloWorldService.Client client = new HelloWorldService.Client(protocol);
        // 准备传输
        transport.open();
        // 正式调用接口
        Reponse reponse = client.send(request);
        // 一定要记住关闭
        transport.close();
        HelloClient.LOGGER.info("response = " + reponse);
    }
}

```

```

    }
    .....
}

```

Apache Thrift 客户端所使用的网络 I/O 模型，必须要与 Apache Thrift 服务器端所使用的网络 I/O 模型一致。也就是说服务器端如果使用的是阻塞式同步 I/O 模型，那么客户端就必须使用阻塞式同步 I/O 模型。

Apache Thrift 客户端所使用的消息封装格式，必须要与 Thrift 服务器端所使用的消息封装格式一致。也就是说服务器端如果使用的是二进制流的消息格式“TBinaryProtocol”，那么客户端就必须同样使用二进制流的消息格式“TBinaryProtocol”。

其他的代码要么就是由 IDL 定义并由 Apache Thrift 的代码生成工具生成，要么就不是重要的代码，所以为了节约篇幅就没有必要再进行赘述了。如图 8-4 所示是运行效果。

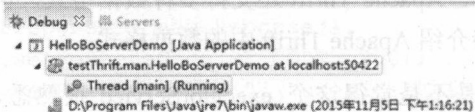


图 8-4 运行效果

- 服务器端收到客户端请求后，取出线程池中的线程进行运行，如图 8-5 所示。

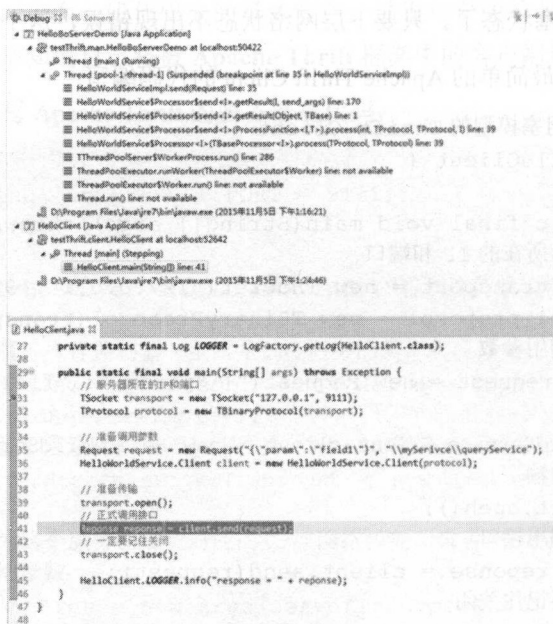


图 8-5 线程池运行效果



## 8.3 RPC 实践：Apache Thrift 深入分析

### 8.3.1 Apache Thrift 与消息格式

Apache Thrift 支持多种消息格式封装。这些消息格式是如何进行编码和解码的，并不需要使用户关心，使用者只需要根据自己的需要指定不同的消息封装格式即可。Apache Thrift 所有消息格式封装的实现，都继承于 TProtocol 这个抽象类，如图 8-6 所示。

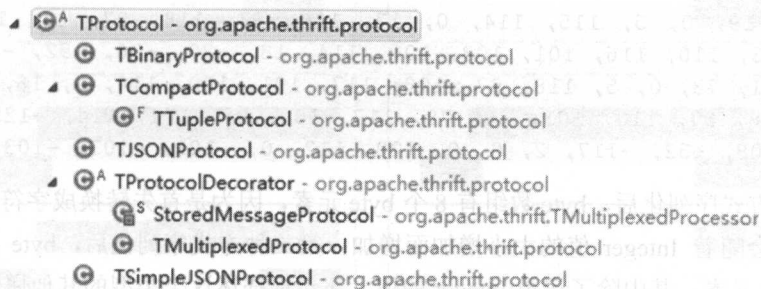


图 8-6 TProtocol 子类结构

#### 1. TBinaryProtocol

TBinaryProtocol 二进制流的编码格式。由于需要支持跨语言特征，所以 Apache Thrift 支持有限的几种通用类型，包括基本类型（Float、Double、Integer、Long、String、Short）、集合类型（Map、Set、List），还有 Pojo 类型（实际上就是前两者若干类型的组合形式）。那么这个类所生成的二进制流和传统的 Java 序列化后生成的二进制流有什么样的区别（或者是优势）呢？我们可以通过阅读 TBinaryProtocol 的源代码进行研究。

我们在 TBinaryProtocol 中，对 Integer 的序列化过程进行详细解释，来对比它和 Java 提供的其他几种序列化方式的不同，并找到不同。首先在 Java 中，如果要将一个 Integer 对象通过网络发送出去，要做的第一件事情就是序列化，那么我们常用的序列化方式有两种，如下所示。

- Java 中序列化 Integer 对象的第一种方法：

```
Integer integerObject = 10066329;
integerObject.toString().getBytes();
```

- Java 中序列化 Integer 对象的第二种方法：

```
ByteArrayOutputStream aStream = new ByteArrayOutputStream();
ObjectOutputStream oStream = new ObjectOutputStream(aStream);
oStream.writeObject(integerObject);
aStream.toByteArray();
```



第一种方式是将 Integer 对象中的值序列化；第二种方式，是将 Integer 整个对象序列化。这两种方式虽然都产生 byte[]，实际上性质是完全不一样的。我们来看一下这两种方式产生的 byte[] 的内容。

- 序列化 Integer 的值：

```
[49, 48, 48, 54, 54, 51, 50, 57]
```

- 序列化整个 Integer 对象：

```
[-84, -19, 0, 5, 115, 114, 0, 17, 106, 97, 118, 97, 46, 108, 97, 110,
103, 46, 73, 110, 116, 101, 103, 101, 114, 18, -30, -96, -92, -9, -127, -121,
56, 2, 0, 1, 73, 0, 5, 118, 97, 108, 117, 101, 120, 114, 0, 16, 106, 97, 118,
97, 46, 108, 97, 110, 103, 46, 78, 117, 109, 98, 101, 114, -122, -84, -107,
29, 11, -108, -32, -117, 2, 0, 0, 120, 112, 0, -103, -103, -103]
```

第一种方式序列化后，byte 数组有 8 个 byte 元素，因为是首先转换成字符串的，所以实际上这个大小会随着 Integer 值的大小增加而增加；第二种方式序列化后，byte 数组中一共有大于 20 个 byte 元素，其中除了记录 Integer 的值，还包括描述这个类型的其他属性。

那么再来看看在 TBinaryProtocol 中，是如何序列化 Integer 类型的。以下代码是 TBinaryProtocol 类中进行 Integer 序列化的源代码，如下所示：

```
.....
private byte[] i32out = new byte[4];
public void writeI32(int i32) throws TException {
    i32out[0] = (byte)(0xff & (i32 >> 24));
    i32out[1] = (byte)(0xff & (i32 >> 16));
    i32out[2] = (byte)(0xff & (i32 >> 8));
    i32out[3] = (byte)(0xff & (i32));
    trans_.write(i32out, 0, 4);
}
.....
```

这个序列化过程实际就是 4 次位运算，我们可以用图 8-7 表示这个序列化过程。

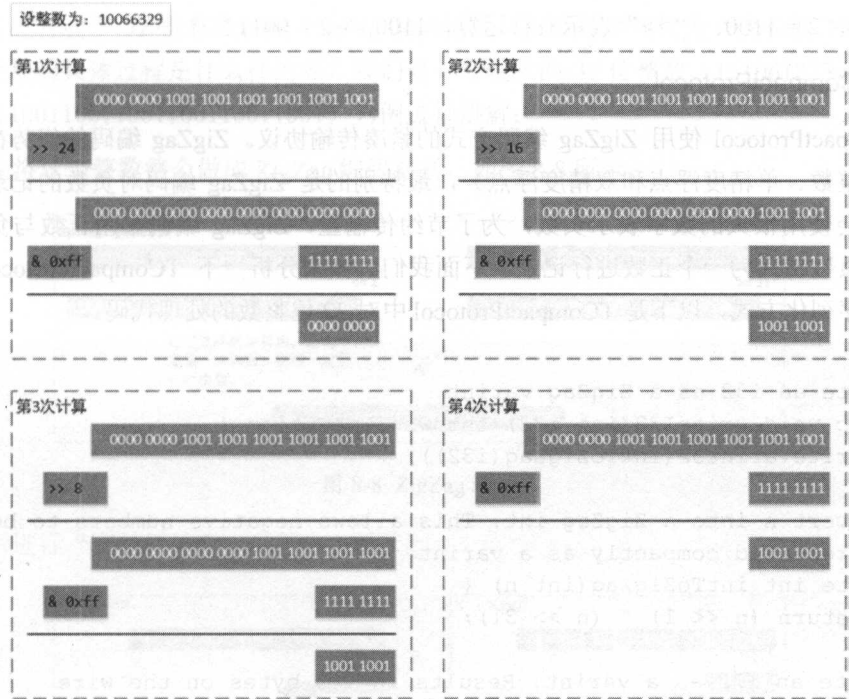


图 8-7 Integer 序列化运算过程

通过 4 次位运算，得到长度为 4 个 byte 数组，并且这个数组的大小并不会随着整数大小的增加而变化。另外位运算的计算速度还是所有计算方式中最快的一种。反序列化的过程相似，对这个大小为 4 的 byte 数组重新进行位计算即可：

```
.....
((buf[off] & 0xff) << 24) |
((buf[off+1] & 0xff) << 16) |
((buf[off+2] & 0xff) << 8) |
((buf[off+3] & 0xff));
.....
```

由于本书篇幅和写作目的所限，不能一一介绍 TBinaryProtocol 的各种序列化方式，但是通过对 TBinaryProtocol 中 Integer 的序列化过程，我们可以找到 TBinaryProtocol 处理过程的优势，包括速度和大小的优势。所以，如果使用环境对序列化过程没有特别的要求，那么直接使用 TBinaryProtocol 进行数据格式的封装就可以了。

这里说明一下：byte 是一个 8 位二进制描述（一个字节），一个 int 需要 4 个 byte 进行表示，如果“0x”的前缀表示 16 进制数字，那么 0xff 的二进制表示就是 1111 1111。“&”是“与”运算符，这个运算符用于二进制计算，1 & 1 = 1，其余情况都 = 0；“<<”表示左移运

算,  $0011 \ll 2 = 1100$ ; “ $\gg$ ”表示右移运算,  $1100 \gg 2 = 0011$ 。

## 2. TCompactProtocol

TCompactProtocol 使用 ZigZag 编码方式的紧凑传输协议。ZigZag 编码的优势在于记录数字类型（整数、单精度浮点和双精度浮点），最特别的是 ZigZag 编码对负数的记录。在计算机中，都会使用很大的数字表示负数，为了节约传输量，ZigZag 编码采用正数与负数交错的方式，把负数转换为一个正数进行记录。下面我们具体来分析一下 TCompactProtocol 中对 32 位整数的序列化方式，以下是 TCompactProtocol 中对 32 位整数的处理代码：

```
.....
//Write an i32 as a ZigZag varint
public void writeI32(int i32) throws TException {
    writeVarint32(intToZigZag(i32));
}
//Convert n into a ZigZag int. This allows negative numbers to be
//represented compactly as a varint
private int intToZigZag(int n) {
    return (n << 1) ^ (n >> 31);
}
//Write an i32 as a varint. Results in 1-5 bytes on the wire
//TODO: make a permanent buffer like writeVarint64
//上面的 TODO 不是本书作者加的
byte[] i32buf = new byte[5];
private void writeVarint32(int n) throws TException {
    int idx = 0;
    while (true) {
        if ((n & ~0x7F) == 0) {
            i32buf[idx++] = (byte)n;
            // writeByteDirect((byte)n)
            break;
        } else {
            i32buf[idx++] = (byte)((n & 0x7F) | 0x80);
            // writeByteDirect((byte)((n & 0x7F) | 0x80))
            n >>= 7;
        }
    }
    trans_.write(i32buf, 0, idx);
}
.....
```

以上代码片段一共只有一个对外的调用方法，另外两个分别名为 `intToZigZag` 和 `writeVarint32` 的方法都是私有方法。从字面上的意义我们可以知道：当对一个 32 位整数进行

编码时，首先将这个 32 位整数转成 ZigZag 编码格式，然后再序列化为“变长的 32 位整数”。那么这个处理的具体过程是什么样的呢？我们以一个较大的 32 位整数（161061273，二进制计数为：1001100110011001100110011001）为例进行讲解：

- 首先将这个整数整个做成 ZigZag 编码格式，如图 8-8 所示。

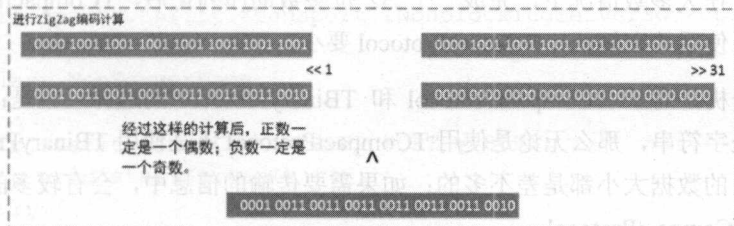


图 8-8 ZigZag 编码

- 然后进行“变长”处理，如图 8-9 所示。

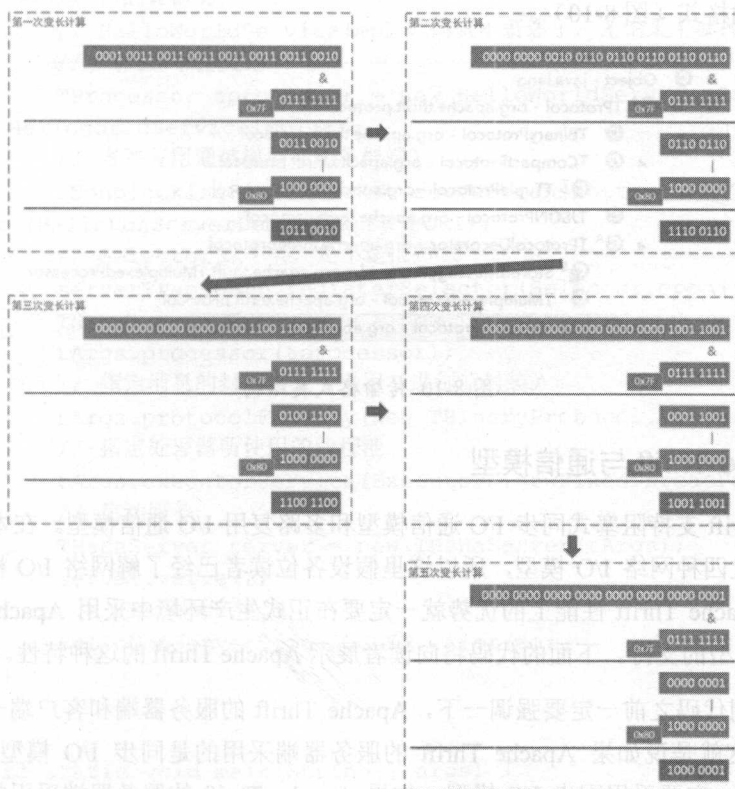


图 8-9 编码变长

可以看到上面的“变长”计算一共进行了 5 次，计算出来的结果比 TBinaryProtocol 中的 32 位整数序列化还要多出一个 byte。这是因为这个数字比较长。

但实际处理中，我们一般使用的数据都是比较小的。这也是为什么首先要使用 ZigZag 编码把某个负数的符号位从高位移动到低位。在实际过程中，变长计算一般只会进行二至三次就完成了。这样，在大多数情况下，完成一个 32 位整数的序列化后，TCompactProtocol 存储序列化结果所需要使用的空间就比 TBinaryProtocol 要小。

那么经过分析，对于 TCompactProtocol 和 TBinaryProtocol 的选择经验是：如果传输的信息中，基本都是字符串，那么无论是使用 TCompactProtocol 还是使用 TBinaryProtocol，其序列化性能和完成后的数据大小都是差不多的；如果需要传输的信息中，会有较多的“低位数字”，那么建议使用 TCompactProtocol。

### 3. 其他传输格式封装

当然，Apache Thrift 还提供其他的传输格式封装。不同的需求场景下，可以根据需要选用这些信息的传输格式（图 8-10）。

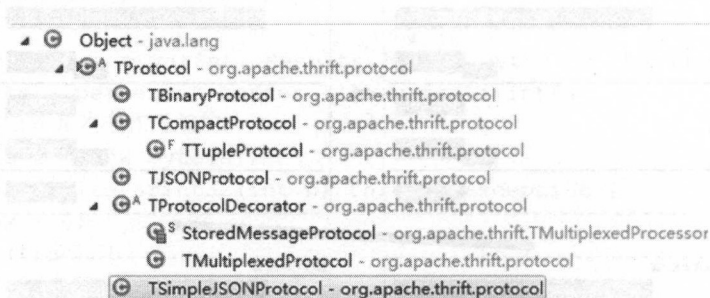


图 8-10 传输格式类结构

## 8.3.2 Apache Thrift 与通信模型

Apache Thrift 支持阻塞式同步 I/O 通信模型和多路复用 I/O 通信模型。在本书第 7 章中较为详细地讨论过四种网络 I/O 模型，所以这里假设各位读者已经了解网络 I/O 模型这个知识点了。要发挥 Apache Thrift 性能上的优势就一定要在正式生产环境中采用 Apache Thrift 对多路复用 I/O 通信模型的支持。下面的代码将向读者展示 Apache Thrift 的这种特性。

在给出示例代码之前一定要强调一下，Apache Thrift 的服务器端和客户端一定要采用相同的通信模型。这就是说如果 Apache Thrift 的服务器端采用的是同步 I/O 模型，那么 Apache Thrift 客户端也一定要采用同步 I/O 模型，如果 Apache Thrift 的服务器端采用的是异步 I/O 模型，那么 Apache Thrift 客户端也一定要采用异步 I/O 模型，否则就无法通信。



- 服务器端的异步 I/O 模型代码:

```

.....
import org.apache.log4j.BasicConfigurator;
import org.apache.thrift.TProcessor;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.server.THsHaServer;
import org.apache.thrift.transport.TNonblockingServerSocket;
.....
public class NonServerDemo {
    .....
    public static final int SERVER_PORT = 8090;
    public void startServer() {
        try {
            // log4j 日志, 如果工程里面没有加入 log4j 的支持, 请替换为 system.out
            HelloNonServerDemo.LOGGER.info("HelloWorld TSimpleServer
start ....");
            // 服务执行控制器 (告诉 Thrift, 实现了 HelloWorldService. Iface 接
            // 口的具体类)
            // HelloWorldServiceImpl 代码就不赘述了, 无论采用哪种通信模型, 它的
            // 代码都不会变化
            TProcessor tprocessor = new HelloWorldService.Processor
<Iface>(new HelloWorldServiceImpl());
            // 多路复用通信模型 (服务器端)
            TNonblockingServerSocket serverTransport = new Thonblocking
ServerSocket (HelloNonServerDemo.SERVER_PORT);
            // Selector 这个类, 是不是很熟悉
            serverTransport.registerSelector(Selector.open());
            THsHaServer.Args tArgs = new THsHaServer.Args(serverTransport);
            tArgs.processor(tprocessor);
            // 指定消息的封装格式 (采用二进制流封装)
            tArgs.protocolFactory(new TBinaryProtocol.Factory());
            // 指定处理器所使用的线程池
            tArgs.executorService(Executors.newFixedThreadPool(100));
            // 启动服务
            THsHaServer server = new THsHaServer(tArgs);
            server.serve();
        } catch (Exception e) {
            HelloNonServerDemo.LOGGER.error(e);
        }
    }
    .....
    public static void main(String[] args) {
        NonServerDemo server = new NonServerDemo();
        server.startServer();
    }
}

```

```

    }
}
.....

```

• 客户端的代码:

```

.....
import org.apache.thrift.TException;
import org.apache.thrift.async.AsyncMethodCallback;
import org.apache.thrift.async.TAsyncClientManager;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TNonblockingSocket;
.....
public class HelloNonClient {
    .....
    private static Object WAITOBJECT = new Object();
    public static final void main(String[] args) throws Exception {
        TNonblockingSocket transport = new TNonblockingSocket("127.0.0.1",
8090);

        TAsyncClientManager clientManager = new TAsyncClientManager();
        // 准备调用参数(这个testThrift.iface.Request是我们通过IDL定义,并且生
        // 成的)
        Request request = new Request("{\"param\":\"field1\"}", "\\
myService \\queryService");
        // 这是客户端对异步网络通信模型方式的支持
        // 注意使用的消息封装格式,一定要和服务器端使用的一致
        HelloWorldService.AsyncClient asyncClient =
            new HelloWorldService.AsyncClient.Factory(clientManager, new
TBinaryProtocol.Factory()).getAsyncClient(transport);
        // 既然是异步 I/O 模型,所以客户端一定是通过“事件回调”方式
        // 接收到服务器的响应通知的
        asyncClient.send(request, new
AsyncMethodCallback<AsyncClient.send_call>() {
            // 当服务器正确响应了客户端的请求后,这个事件被触发
            @Override
            public void onComplete(send_call call) {
                Reponse response = null;
                try {
                    response = call.getResult();
                } catch (TException e) {
                    HelloNonClient.LOGGER.error(e);
                    return;
                }
                HelloNonClient.LOGGER.info("response = " + response);
            }
        })
    }
}

```

```

// 当服务器没有正确响应客户端的请求
// 或者其过程中出现了不可控制的情况时，这个事件会被触发
@Override
public void onError(Exception exception) {
    HelloNonClient.LOGGER.info("exception = " + exception);
}
});
//这段代码保证客户端在得到服务器回复前，应用程序本身不会终止
synchronized (HelloNonClient.WAITOBJECT) {
    HelloNonClient.WAITOBJECT.wait();
}
}
}
.....

```

以上代码是可以直接工作的，读者可以直接在自己的工程中执行。目前各种主流的 RPC 框架基本都支持多路复用 I/O 模型，如果你有兴趣进行这些 RPC 框架的性能比较，那么一定要在相同的 I/O 通信模型下进行。

### 8.3.3 Apache Thrift 与线程池

在本章第 8.1 节，我们已经提到影响一款 RPC 框架性能的主要指标。除了 RPC 框架实现的数据封装格式、RPC 框架支持的网络通信模型，还有一个重要的指标就是它如何执行客户端的请求。例如在 Apache Thrift 中，无论 Apache Thrift 使用哪种数据封装格式，使用哪种网络通信模型，都使用线程池技术运行具体的接口实现，响应客户端请求，

```

org.apache.thrift.server.THSaServer.Args.executorService(ExecutorService executorService)

```

可以看到，实际上在 Apache Thrift 中设置线程池的方法，所要求的参数类型是 `java.util.concurrent.ExecutorService` 接口的实现类，也就是说只要实现了 `ExecutorService` 接口的类都可以被传入。一般我们常使用的是 `java.util.concurrent.ThreadPoolExecutor` 这个类。如果你还不了解 Java 中线程池的相关知识，可以参考本书第 7 章第 7.8 节中的内容。除了 `ThreadPoolExecutor` 这个实现类，在 JDK 1.7+ 中还提供了新的 Fork/Join 框架，也可以使用。

## 8.4 RPC 实践：解决异常问题

Apache Thrift 是一个完成系统间调用的好选择，但它也不能解决系统间调用的所有问题。试想一下这样的场景，有三个系统使用 Apache Thrift 参与的系统间的调用过程：A 系统对 B、C 两个系统进行调用，正常的业务要求是只有 B 系统和 C 系统提供的接口都调用成功后，才认

为整个业务成功了，否则认为整体业务失败。

这时候使用 Apache Thrift 进行 RPC 调用就出现了一个看似无法解决的问题：当 A 系统调用 B 系统的业务接口成功后，前者对 C 系统提供的业务接口进行的调用可能失败，这个时候 B 系统中已经变更的数据就需要被回滚。而 C 系统提供的 Apache Thrift RPC 接口虽然定义了异常，可以在 C 系统的业务接口调用失败后，向调用者抛出异常，但却无法在 C 系统调用失败后通知 B 系统进行回滚（图 8-11）。

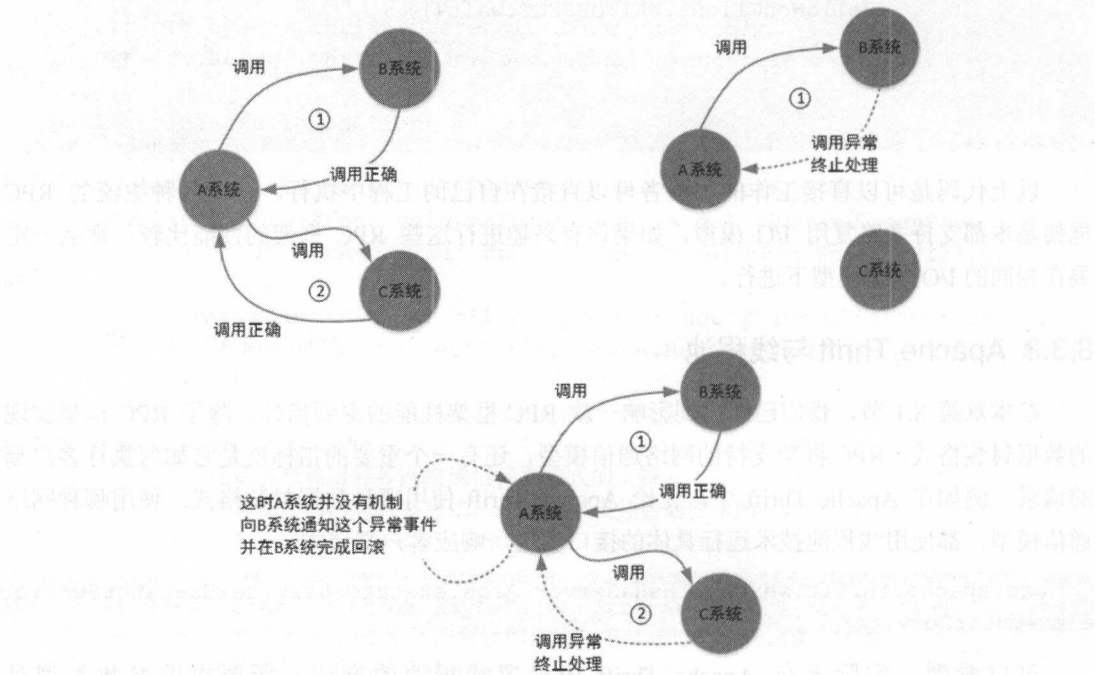


图 8-11 异常问题的产生

是的，我们正在讨论一个分布式异常的问题。此类问题普遍存在于两个或者多个系统间服务层接口调用的环境下，只要整个业务过程涉及两个以上原子服务就可能会有这个问题。本节将重点讨论如何解决这类问题，首先将介绍目前行业中解决此类问题的几种流行方案和注意事项。然后给出一个基于命令模式实现多服务协调的示例，包括将 Netflix Hystrix 集成进来一同使用，这个示例可适用于多系统间服务接口规模还不太庞大的业务场景下。最后我们会讨论在系统间接口规模达到一定的复杂程度，并引入了某种服务治理框架后，使用服务治理框架本身的机制或者第三方机制来解决这类问题。



8.4.1 分布式业务的异常解决思路

读者是否在第一时间就想到事务这个概念呢？传统意义上的事务被定义在数据层面（例如各种关系型数据库的实现上），它是指一组原子操作，这组原子操作必须按照既定的顺序全部执行成功。如果某一个或者多个原子操作失败，则回退所有之前的原子操作到原来的状态。事务的特点主要有四个：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。一个标准的事务必须同时满足这四个特性，否则就无法保持业务数据的正确性——并且在 SQL 规范中也明确定义了这些特性的实现标准，例如它定义了四类事务隔离级别：Read Uncommitted（读未提交）、Read Committed（读已提交）、Repeatable Read（可重复读）和 Serializable（可串行化）。

传统业务环境下这些原子数据操作都在同一个数据库实例上完成，而随着企业中各系统的复杂度增加，就可能会出现事务跨两个或者多个系统数据库实例的情况，后一种事务处理机制就是常说的分布式事务。

1. 分布式事务与两阶段提交协议（2PC）

要说明分布式事务的工作原理就要先理解它的实现理论——两阶段提交协议（2PC）。简单来说这个协议中提到两个阶段是指准备阶段和提交阶段（图 8-12）。

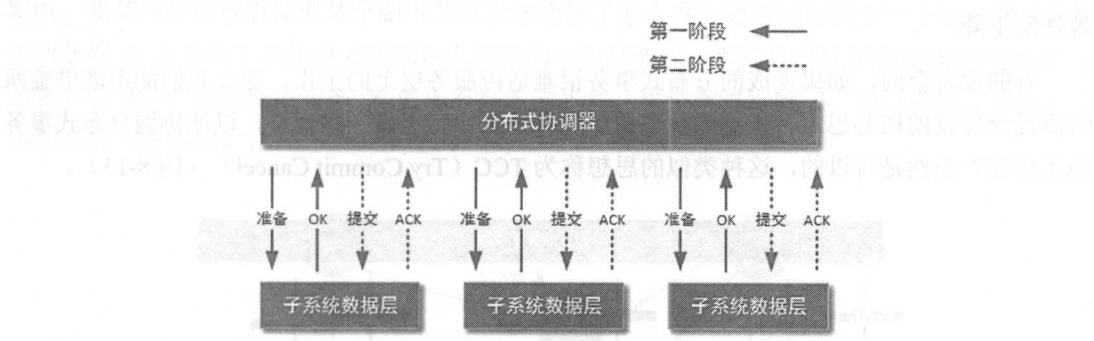


图 8-12 两阶段提交协议（2PC）

除了两阶段提交协议（2PC），它还有一些变体协议，例如三阶段协议。当然两阶段提交协议是最典型的事务协议，它的实现一般需要一个事务协调者来统一所有工作，协调者首先向参与事务动作的各个数据节点提交准备请求（也叫检查请求），并等待所有参与此事务的数据节点返回确认信息。当各个数据节点收到这个准备请求后就会检查自己是否有条件执行自己负责的处理部分，并且执行但不提交各自负责的这部分数据操作。这是一个同步过程，在收到所有节点确认信息之前协调者都不会发出下一步的执行指令，各个节点被锁定/预占的资源也不会被释放——因为各数据节点的处理过程都未正式提交。



如果在这个过程中任何一个数据节点返回了“不可用”或者等待超时，那么协调者就会向各个参与事务动作的处理节点发出“回滚”指令，各个处理节点就会回滚/释放资源，并将回滚结果回馈给协调者；如果协调者收到所有处理节点“准备好”/“同意”的信号，那么就会进入第二个阶段——通知各个处理节点“提交”。当各个处理节点收到第二步指令后，就会正式提交上一步已经完成的处理动作，并向协调者回馈“完成”消息。

分布式事务已经有非常多的实现组件和成功案例，例如第三方组件 JOTM（Java Open Transaction Manager）就是一个成熟的分布式事务管理器，现在许多 Tomcat 服务器上都是用它提供的分布式事务功能；再例如 Atomikos 也是一款常用的分布式事务管理器，也是目前配合 Spring 集成的一款分布式事务管理器。

但是分布式事务从定义和实现上都不适用于我们解决本节开头所描述的问题类型。首先系统间的 RPC 调用工作在系统的服务层，一般来说它们都是无状态，且可以独立完成工作的。而本小节介绍的分布式事务却作用于系统数据层，且这些数据层存在数据依赖性，也就是说它们的耦合性会比较高（无论是历史原因还是业务设计原因）；另外一个关键点是，工作在服务层的各个原子服务在系统中负责的业务意义和数据层所负责的业务意义有本质区别——甚至某个原子服务为了完成一个业务动作还会对数据库进行多次操作。最后，基于两阶段提交协议的分布式事务，虽然可以达到近实时的一致性，但多次同步/检测执行状态所付出的代价就是处理性能下降。

有的读者会问，如果现成的分布式事务很难适应服务层上的工作，那么我们能不能借鉴两阶段提交协议的核心思路，依据无状态原子服务的工作特点做一些调整，以便协调分布式事务的工作呢？当然是可以的，这种类似的思想称为 TCC（Try Commit Cancel）（图 8-13）。

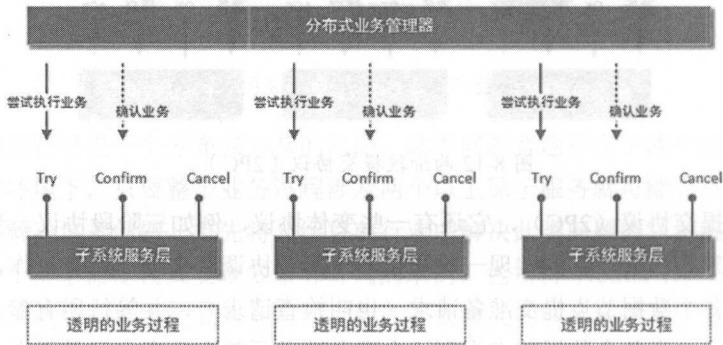


图 8-13 TCC

TCC 过程实际上是 2PC 的一种变形，它与后者的关键区别在于 TCC 的思路着重于原子服务而非具体数据操作。Try 步骤的细分工作又包括业务检查和资源预占，而对于如何实现这个

过程，在 TCC 中并没有明确要求必须通过对持久层数据进行写操作，这就便于架构师在服务层而非在数据层上设计 TCC 的实现。

## 2. BASE

分布式系统的知识体系中有一个非常重要的概念——CAP 原理。这个原理指导着大多数分布式系统的设计过程，CAP 原理大致是说分布式系统中一定存在三个特性：一致性（Consistency）、分区容忍性（Partition）和可用性（Availability）。

举个例子，一个分布式系统中有  $N$  个节点通过网络连接在一起协同工作。首先你不能将完整的数据  $X$  只存放在一个节点上，这是因为一旦这个节点由于各种原因停止工作，数据  $X$  就不能被访问，这样肯定就不再满足系统的持续可用性了，并且一旦这个节点不能再被恢复，数据  $X$  就会永远丢失了。所以数据  $X$  至少也应该在不同的节点上存储多份，存储的副本量越多越能保证数据  $X$  的安全，也越能保证即使在多个节点同时不可用的情况下，数据  $X$  也同样能够被访问。这就是分区性的要求，按照普遍经验，数据  $X$  的副本数至少应该有三份。

那么当数据  $X$  发生变化时如何对这些副本进行更新呢？最理想的效果是，当客户端发出数据  $X$  的更新请求后，从任何一个节点访问数据  $X$  都可以拿到它最新的状态，这就是一致性要求。当然这个最理想的效果太理论化了，要知道基于网络工作的分布式系统受很多外在因素影响：要是同步过程中发现某个副本节点无法连接了怎么办？要是同时又有一个客户要求再次更新数据  $X$  怎么办？如果真要达到这么理论的一致性要求，那就只能让所有需要读/写数据  $X$  的客户端等待，直到完成数据  $X$  的所有副本同步后，再依次进行响应。但这样做又不满足可用性要求（图 8-14）。

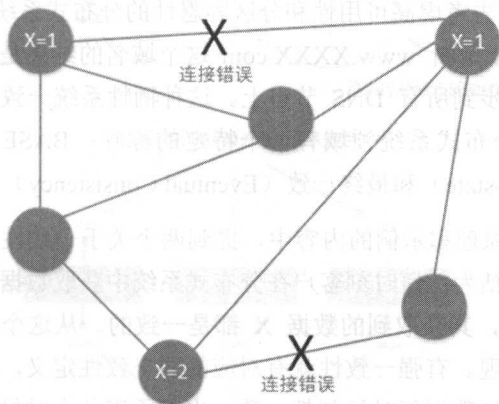


图 8-14 CAP 原理

关系型数据库的设计通常基于 ACID 原理，即原子性（Atomicity）、一致性（Consistency）、

隔离性 (Isolation) 和持久性 (Durability)，而且关系型数据库普遍采用事务技术实现 ACID 原理，其中每一个事务就是最小的原子操作，还可以设置不同的事务级别，包括可读未提交、可重复读这样的事务级别；关系型数据库还规定了一旦事务正确提交就不能进行数据回滚，如果还要继续修改数据就只能启动一个新的事务。而这一切都是为了保证数据库系统是一个强一致性系统。就连架设在各个关系型数据库实例之上的分布式事务机制，也是为了保证这个目标：只要有任何一个参与分布式事务过程的数据库实例出现异常，整个分布式事务就无法正常提交，当然也就无法完成数据写操作。注意，数据是可以在事务操作的同时进行读操作的，即使某些节点出现了问题其他数据库节点也可以承担这个读操作，因为这样的操作不存在一致性改变的风险。

任何分布式系统都不可能以 CAP 原理中三个特性同时作为设计目标，要达到非常高的强一致性和较高的分区容忍性，就必须以牺牲可用性作为代价（注意是牺牲而不是完全放弃）。而分区容忍性又是分布式系统成立的基础，没有任何分区容忍性的分布式系统甚至都不能称为分布式系统。高压力的环境下又不能过度牺牲分布式系统的可用性，要知道 99.99% 的可用性和 99.999% 的可用性完全就是两个档次的分布式系统。

所以类似分布式事务机制那样过度强调数据一致性的设计思路，在分布式系统的设计过程中就不太受主流思想欢迎，至少从目前各种分布式系统公布的设计原理来看是这样的。例如 HDFS 这样的分布式文件系统，首先强调的是高可用性和一定的分区容忍性，其次才是数据一致性，数据一致性通过副本保证，但并不是所有副本都完成写入动作后 HDFS 才认为数据一致，而是只要一部分数据副本完成了写入动作，HDFS 就认为数据成功写入且客户端可以调用新数据，而没有完成同步的副本将会接着进行数据同步，达到数据的最终一致性。

DNS 也是一种需要首先考虑高可用性和分区容忍性的分布式系统，DNS 的组织结构只能保证多个 DNS 服务节点解析 `www.XXXX.com` 这个域名的结果是最终一致的而不能保证新的域名映射关系被立即同步到所有 DNS 节点上。这种牺牲系统一致性保证系统可靠性和分区容忍性的设计思路，在分布式系统领域有一个特定的称呼：BASE，即基本可用 (Basically Available)、软状态 (Soft-state) 和最终一致 (Eventual Consistency)。

以上简单说明 CAP 原理和示例的内容中，提到两个关于一致性的概念：强一致性和最终一致性。强一致性可以概括为任何时刻客户在分布式系统中获取数据 X，无论它在分布式系统的哪个节点进行这个操作，其获取到的数据 X 都是一致的。从这个定义来看，分布式事务机制就是一种强一致性的实现。有强一致性就有对应的弱一致性定义，弱一致性不是说不保持数据的一致性，而是说不保证数据每时每刻都一致，也不承诺什么时候才能保证分布式系统的任何节点都能读取到一致的数据。而最终一致性是弱一致性的一种特定结果，即是承诺基于弱一致性，在经过一个数据不一致的时间窗口后，最终能保证数据一致。这个数据不一致的时间窗

口在客户端看来非常短，而且分布式系统还可以通过多种方式向客户端屏蔽不一致的数据，例如主从副本方式。

虽然本节讨论的跨系统的多服务调用的场景中，不能将“跨系统服务调用”称为一个分布式系统——它不具备功能内聚性，不能称之为系统，最多只能算原子服务存在分布性。但是我们可以从本小节概要介绍的分布式系统设计原理，找到一些解决问题的启发。BASE 的核心思路是在保证可用性和分区容忍性的前提下，找到一个牺牲一致性的程度，而最终一致性为我们指明了这个程度——保证数据最终一致即可。那么对于跨系统分布式业务我们也可以借鉴这个概念：各个原子服务无须关心其他原子服务什么时候执行或者执行顺序，只需要各自负责自己的那部分业务，并最终完成处理即可。如果出现需要回退的情况，则各个原子服务负责取消自己负责的那部分操作，回到原始状态。

### 3. 事务补偿机制

这种机制本身并不提供事务，而是在需要进行回滚操作时能够依据某种手段获知整个执行路径，并完成符合业务规则的逆向执行动作。为了保证事务补偿机制能够运行，业务系统提供了某个原子服务就必须提供一个和这个原子服务反向操作的另一个原子服务，这样才可以保证事务补偿机制在任何一个正向执行的分布式业务工作的环境中，都有一个与之对应的反向执行过程（图 8-15）。

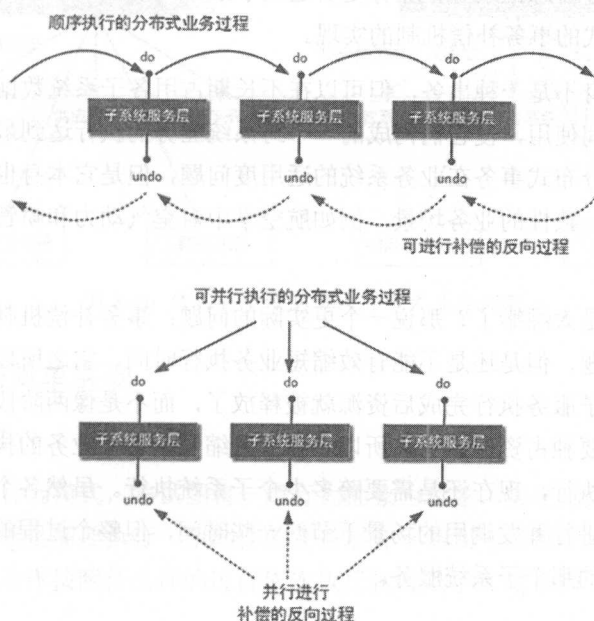


图 8-15 事务补偿机制

试想一下这样的场景，一个需要带有事务性的分布式业务过程，其执行步骤涉及多个子系统的原子服务。这些原子服务耦合性都很低，脱离其他原子服务后也可以独立工作，并且整个业务过程持续时间较长（例如一般情况下也需要 1 秒才能完成一个正常执行过程）。这种情况下如果基于某个分布式事务组件进行业务过程控制不但会造成分布式协调器过重的负担，而且由于两阶段提交协议的工作方式，在数据层占用的资源也一直得不到释放，这又增加了单个系统数据层的负担。如果这个跨系统的业务服务并发规模不大，那么还不至于对整个顶层设计造成影响，但如果达到一定的并发规模，就将对整个系统的顶层设计造成极大的负面影响。

支持事务补偿机制的分布式业务不会存在这个问题，当一个原子服务被执行时数据即时被更改，占用资源使用后即时被释放，执行日志被详细记录。如果某一个原子服务执行失败并不是将之前未提交数据回滚，而是通过一个对应的反向服务将之前的结果逆向执行，这里的逆向执行将重新修改数据、重新占用资源、重新生成新的日志。根据各个原子服务间的依赖性，这个执行过程既可以是顺序执行的又可以是并行执行的。

事务补偿机制可以通过多种方式进行实现，例如独立实现一个事务补偿控制模块，并向这个模块注册每一个正向方法和对应的逆向方法，跨系统业务由这个模块负责执行并在出现执行错误的情况下进行重试或者逆向执行；还可以采用日志方式进行实现，将跨系统业务的发起方和各个执行方执行的每一步写入日志，并在出现执行错误时利用日志记录的信息分析和执行回滚策略，最后还要通过日志记录整个业务是否达到了最终一致性。本节的后续内容中，我们将介绍一种基于命令模式的事务补偿机制的实现。

事务补偿机制本身不是一种事务，但可以在不长期占用各子系统数据资源的前提下，在耦合度较低的原子服务间使用，使它们构成的一个跨系统业务的执行达到最终一致性。事务补偿机制虽然部分解决了分布式事务在业务系统的适用度问题，但是它本身也不是完美的。例如它不适用需要保证实时一致性的业务场景，例如航空学中对空气动力和喷管尾焰进行分析的强实时性系统。

上面的例子是不是太缥缈了？那说一个更实际的问题：事务补偿机制可以解决长耗时业务过程中资源占用的问题，但是还是不能有效缩短业务执行时间。它之所以能够解决资源占用问题，是因为在每个原子服务执行完成后资源就被释放了，而不是像两阶段提交协议那样在整个业务执行完成前都需要独占资源。它之所以不能有效缩短一个长业务的执行时间，是因为之前需要跨多少个子系统执行，现在还是需要跨多少个子系统执行。虽然各个业务系统原子服务没有依赖关系，可以在进行并发调用的场景下节约一些时间，但整个过程的用时还是取决于这些原子服务中耗时最长的那个子系统服务。



#### 4. 可靠的异步消息

在一个电商系统中，从消费者视角来看怎样才算一个订单完成了呢？显然是这个订单所涉及的所有商品送到消费者手中哪一刻起。但从订单付款到送货上门的时间周期非常长，短则一天长则可能是一个月。而从电商系统的角度来看，这个过程可能涉及了很多系统，例如付款子系统、供应商系统、库存子系统、物流配送系统和客服系统。难道要电商系统等待这些子系统全部返回调用结果，才算作一个订单业务完成吗？显然这是不行的，也是不符合一般规律的。看来我们需要寻找一种两阶段事务和事务补偿机制以外的处理方式来解决这类问题。

可靠的异步消息系统是目前解决这类问题所常用的方式，“可靠”的定义是消息系统不会无故丢失消息，保证消息送达到指定的目标系统；“异步”是指消息的发送方（又称生产方）在送出消息后，无须等待消息接收方（又称消费方）反馈任何结果即可执行后续的操作。典型的可靠的异步消息系统有 ActiveMQ、RabbitMQ 和 RocketMQ，在本书的第 9 章中我们将详细讨论消息队列系统的使用和性能优化。图 8-16 给出了电商系统中使用消息系统解决该类问题的顶层架构思路。

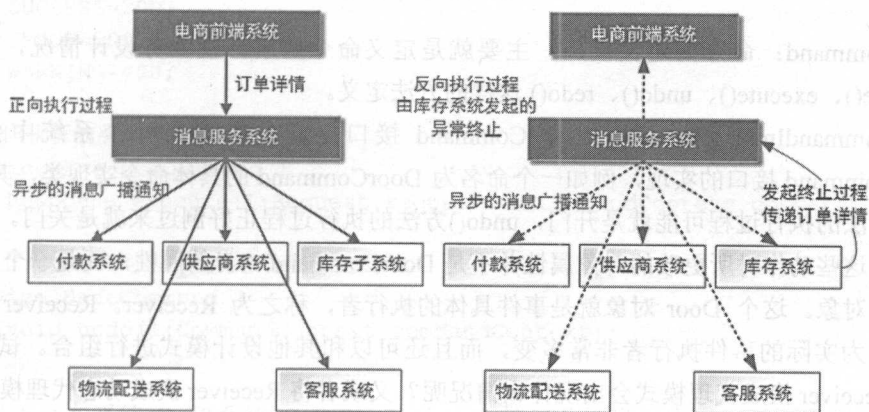


图 8-16 使用异步消息解决分布式业务问题

### 8.4.2 事务补偿的简单实现

#### 1. 命令模式原理

命令模式是一种行为模式，它试图将一组行为抽象为对象，实现行为请求者和行为实现者的松耦合。具体来说就是需要执行一组动作的业务层并不需要知道每个业务动作是怎样执行的，只需要清楚每个业务动作按照什么样的过程依次执行就可以了（图 8-17）。

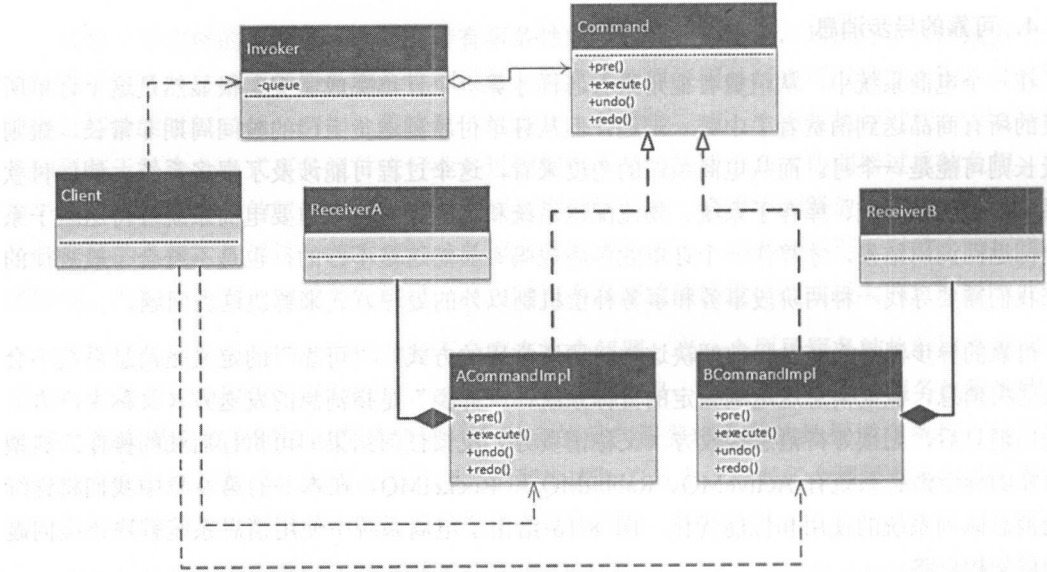


图 8-17 命令模式

- **Command**: 命令的定义接口，主要就是定义命令要素。视业务设计情况，一般会有 `pre()`、`execute()`、`undo()`、`redo()` 这样的方法定义。
- **CommandImpl、Receiver**: 对 **Command** 接口定义的实现，一个系统中会有多个 **Command** 接口的实现，例如一个命名为 `DoorCommand` 的具体命令实现类，其 `execute()` 方法的执行过程可能就是开门，`undo()` 方法的执行过程正好倒过来就是关门。注意，执行这些动作时所更改的对象属性并不是 `DoorCommand` 本身的属性，而是一个名叫 `Door` 的对象。这个 `Door` 对象就是事件具体的执行者，称之为 **Receiver**。**Receiver** 非常重要，因为实际的事件执行者非常多变，而且还可以和其他设计模式进行组合。试想如果将 **Receiver** 换成代理模式会出现什么情况呢？又或者将 **Receiver** 换成动态代理模式呢？
- **Invoker**: **Invoker** 用来编排和协调每个原子服务动作的执行过程，其中一般会有一个用来存储将要执行的多个命令的一个集合。并按照上一个业务动作的执行情况，确定如何执行下一个业务动作。（是不是特别像责任链模式的工作描述呢？）

## 2. 顺序执行原子服务

如果参与分布式业务的各个原子服务间存在业务依赖，那么就需要依次执行这些原子服务。在这个顺序执行的分布式业务示例中，我们将使用本章介绍的 **Apache Thrift** 来作为命令模式中各个跨系统服务间调用的基础，由于 **Apache Thrift** 有自己的 IDL 和 Proxy 生成过程，所以我们需要将命令模式的 **Command** 接口在 IDL 中进行定义，并在提供这些服务的系统上进行实现。

- 以下是 IDL 定义:

```
namespace java test.command.rpc
# 这个结构体用于描述每个原子服务调用时的各种参数传入
struct Request {
    1:required string paramValue;
    2:required string paramname1;
}
# 这个结构体用于描述从原子服务返回的调用结果
struct Response {
    1:required RESCODE responseCode;
    2:required string responseValue;
}
# 这里的信息保证多次相关联的远程调用信息能够被联系起来
struct CommandContext {
    1:required string commandUid;
}
# 这个枚举表示返回信息的类型
enum RESCODE {
    SUCCESS=200;
    ERROR=500;
    WARNING=400;
}
# 正向执行的服务定义
service ExecuteCommand {
    Response execute(1:Request request, 2:CommandContext commandContext);
}
# 与正向执行相对应的逆向执行服务定义
service UndoCommand {
    void undo(1:CommandContext commandContext);
}
```

- 以下是在系统 B 和系统 C 中对 Command 接口分别进行实现，而不同系统中不同的 Command 实现都不一样。由于是一个示例过程，所以系统 B 和系统 C 对正向服务、逆向服务的实现基本一致，这里只给出系统 B 的实现，系统 C 的实现就不再赘述了：

```
.....
// 这是系统 B 对正向调用定义的实现
public class FirstExecuteCommand implements test.command.rpc.ExecuteCommand.
Iface {
    .....
    @Override
    public Response execute(Request request, CommandContext command
Context) throws TException {
        // 这是正式的处理过程。返回信息中带有对处理结果的描述
```

```

// 在本示例中，错误信息也是携带在 Response 中进行返回的，
// 但实际过程中可以通过抛出异常的方式进行通知
Response response = new Response();
// 为了模拟调用出错的情况
// 这里限定随机 10% 的调用出错率，20% 的几率出现警告，10% 的几率直接抛出异常
float random = new Random().nextFloat();
// 如果以下条件成立，则说明调用失败，否则认为调用成功
if(random <= 0.1f) {
    response.setResponseCode(RESCODE.ERROR);
}
// 如果条件成立，则说明调用过程出现警告
else if(random > 0.1f && random <= 0.3f) {
    response.setResponseCode(RESCODE.WARNING);
}
// 如果条件成立，则直接抛出异常
else if(random > 0.3f && random <= 0.4f) {
    throw new TException("FirstRemoteCommand execute
exception !");
} else {
    response.setResponseCode(RESCODE.SUCCESS);
}
response.setResponseValue(new Date().toString());
log.info("FirstRemoteCommand execute , command uid : " +
commandContext.getCommandUid() + " , response code is " + response.
getResponseCode());
return response;
}
}

// 这是系统 B 对逆向服务定义的实现
public class FirstUndoCommand implements test.command.rpc.UndoCommand.
Iface {
    .....
    @Override
    public void undo(CommandContext commandContext) throws TException {
        // 单个服务节点在整个事物出错后，通过这个方法进行业务数据回滚
        // 在正式过程中，可以通过传入的参数描述错误的具体情况
        // 以便识别不同情况的回滚情况
        log.info("FirstRemoteCommand command uid: " + commandContext.
getCommandUid() + " now undo!");
    }
}
}

```

- 进行分布式业务顺序执行的 Invoker 在系统 A 中进行实现，并且在其中加入 Command



命令集合的管理。其中的关键就是对所有用到的 Command 步骤进行编排和顺序执行。为了便于演示，这里我们使用 JUnit 单元测试的方式，顺序调用系统 B 和系统 C 对业务的正向执行过程：

```

.....
// 测试前的准备工作
@Before
public void preExecute() {
    // 第一个远程服务的地址
    RemoteInfo remoteInfo = new RemoteInfo();
    remoteInfo.setIp("127.0.0.1");
    remoteInfo.setPort(8081);
    remoteList.add(remoteInfo);

    // 第二个远程服务的地址
    remoteInfo = new RemoteInfo();
    remoteInfo.setIp("127.0.0.1");
    remoteInfo.setPort(8082);
    remoteList.add(remoteInfo);
}
@Test
public void execute() {
    // 整个调用过程如下
    // 1. 根据 remoteList 中已经确定的远程地址（实际上就是已经确定的调用过程），
    // 完成所有远程连接的初始化
    // 并确定本次调用的唯一 id 信息，存储到 commandContext 上下文中
    // 如果有某一个服务的远程连接无法打开，那么肯定整个事务就无法处理了，
    // 这种情况直接退出
    // 2. 依次执行所有远程接口的 execute 方法，
    // 并依靠 commandContext 上下文实现各接口间的数据传递
    // 2.1. 完成上一个接口调用后，确定是否执行成功
    // 2.2. 如果执行成功或者只是出现了警告，则进行下一个远程接口的调用，
    // 否则执行第 4 步。
    // 3. 如果在调用过程中，某一个远程接口返回了错误信息，则说明整个调用失败
    // 这时向上回滚通知已经完成调用的接口，执行 undo 方法
    // 并依靠 commandContext 上下文实现各接口间的数据传递
    // 4. 无论每一个远程的调用接口执行结果如何，也无论整个事务的执行情况如何
    // 远程链接都要进行关闭
    int index = 0;
    // 1. =====
    try {
        for (; index < remoteList.size(); index++) {
            RemoteInfo remoteInfo = remoteList.get(index);
            this.openRemoteConnect(remoteInfo);

```



```

    }
    } catch (TTransportException e) {
        log.error(e.getMessage(), e);
        // 关闭已经成功打开的远程链接
        this.closeRemoteConnect(remoteList, index);
        return;
    }
    // 这里建立 Command 上下文, 目前里面只有一个唯一编号的 uid 信息
    CommandContext commandContext = new CommandContext();
    commandContext.setCommandUid(UUID.randomUUID().toString());
    // 这里开始构建 request 请求信息,
    // 也有可能每个服务节点的处理过程使用的 request 是不一样的
    Request request = new Request();
    request.setParamname("paramname");
    request.setParamValue("paramvalue");
    boolean transactionSuccess = true;
    //2. =====
    for(index = 0; index < remoteList.size(); index++) {
        RemoteInfo remoteInfo = remoteList.get(index);
        ExecuteCommand.Client client = (ExecuteCommand.Client)
remoteInfo.getExecuteClient();
        try {
            Response response = client.execute(request, command
Context);

            // 2.1=====
            // 如果条件成立, 则说明执行失败了
            // 但就如本章内容所述, 更科学的方式应该是抛出异常
            if(response == null || response.getResponseCode() ==
RESCODE.ERROR) {
                transactionSuccess = false;
                break;
            }
            // 这里还可以引入监听器模式或者观察者模式
            // 向外部关心执行过程的订阅者返回执行事件
            // 例如: onNextExecute() 事件
            } catch (TException e) {
                log.error(e.getMessage(), e);
                transactionSuccess = false;
                break;
            } finally {
                //TODO 无论正向执行是否成功, 都要进行日志记录
            }
        }
    }
    // 3. =====

```

```

// 如果条件成立, 说明事务执行失败, 则开始进行 undo 执行
for(;!transactionSuccess && index >= 0 ; index--) {
    RemoteInfo remoteInfo = remoteList.get(index);
    UndoCommand.Client client= (UndoCommand.Client )remoteInfo.
getUndoClient();
    try {
        client.undo(commandContext);
    } catch (TException e) {
        log.warn(e.getMessage() , e);
    }
}
//4. =====批量关闭即可
this.closeRemoteConnect(remoteList, remoteList.size() - 1);
}
.....

```

本小节代码演示了命令模式中几个重要元素的实现, 包括 Command 接口定义、接口实现、Receiver 和 Invoker。其中 Command 接口遵循 Apache Thrift 中对接口的定义, 利用 IDL 还可以实现接口定义的跨平台性。另外在本示例中 Receiver 身份就是使用 Apache Thrift 支持的客户端实际进行 Apache Thrift Server 的调用。最后使用单元测试的方式实现了一个 Invoker 角色, 完成对所有命令元素的编排和调用。以下在控制台显示的信息, 是以上代码可能的执行效果:

```

.....
12:41:10.241 [pool-2-thread-2] INFO test.command.server. FirstExecute
Command - FirstRemoteCommand execute , command uid : 21bc891b-6468-4477-
86c5-b52dac96db1f , response code is WARNING
12:41:12.589 [pool-1-thread-2] INFO test.command.server. SecondExecute
Command - SecondRemoteCommand execute , command uid : 21bc891b-6468-4477-
86c5-b52dac96db1f , response code is ERROR
12:41:16.193 [pool-1-thread-2] INFO test.command.server. SecondUndoCommand
- SecondRemoteCommand command uid : 21bc891b-6468-4477-86c5-b52dac96db1f now
undo!
12:41:18.266 [pool-2-thread-2] INFO test.command.server.FirstUndoCommand
- FirstRemoteCommand command uid : 21bc891b-6468-4477-86c5-b52dac96db1f now
undo!
.....

```

但是这样实现的分布式业务控制方式除了前文讨论到的不能有效缩短整个分布式业务的全过程执行时间、不能解除原子服务间不必要的依赖性, 还存在很多问题。而这些问题所涉及的要害无论是成熟但适用场景不佳的分布式事务, 还是为保持最终一致性的事务补偿机制, 或者是可异步工作的可靠消息系统, 在设计和使用时都需要注意以下几点。

- 注意重试操作: 由于分布式业务的各个执行步骤存在于不同的系统中, 这些系统依靠

网络进行互相通信，所以在服务的调用过程中不可能保证 100%成功。也就是说即使每个原子服务工作正常，整个分布式业务的工作过程也可能因为某些工作环境原因而调用失败，所以一旦某个原子服务调用失败，协调者要做的事情并不是立即进行业务回滚、逆向操作或者异常消息通知，而应该首先进行重试操作，这是为了排除包括网络抖动、主从服务切换在内的服务临时不可用情况。经过一定数量的重试后，如果服务还是不可用，则协调者再进行业务回滚、逆向操作或者异常消息通知。以上示例代码的客户端并没有任何的重试过程，就开始进行逆向操作了。

- 注意幂等性：所谓幂等性是指参与同一个分布式业务的各个操作步骤，无论执行多少次操作动作其结果都与执行一次操作动作后的结果一致。实际上这是对重试操作特性的一种支持，由于重试操作在异常情况下进行，所以谁也不能保证原子服务不会被错误地多次调用，甚至不能保证请求发起者不会错误地重复发起同一个分布式业务操作过程。例如，订单生成时要避免重复生成的情况发生、订单执行过程要防止出现重复生成配送单的情况、订单逆向执行的退款步骤要防止重复退款的情况发生。
- 注意基于日志：日志的作用可概括为事中记录和事后回溯，一个分布式服务过程的每一个执行步骤都应该详细记录日志，无论这个过程是正向执行还是逆向执行；在分布式服务执行完成后也应该记录日志，无论这个分布式服务执行是否成功。这些日志可以在分布式业务执行结束后检查最后的执行效果，在协调器出现宕机时利用日志恢复协调器执行状态保持执行过程的一致性，在需要进行逆向执行时也可利用日志找到对应的回溯路径。以之前的示例代码来说，Invoker 角色中的代码虽然可以在某个原子服务出现异常时，按照分布式业务对应的逆向过程进行执行，但是却没有使用任何办法保证逆向过程执行的可靠性——没有记录日志。所以要是某个原子服务执行的逆向过程失败，整个系统的数据一致性就没有保证了。

### 3. 并行执行原子服务

参与分布式服务的各个原子服务，在没有业务间依赖的情况下可以采用并行执行的方式缩短整个分布式服务的执行时间，提高执行效率。以下代码示例将演示这个要点，并且我们还会在以下代码中增加逆向执行失败后的日志记录，以便执行器能够定时检索这些失败日志并在适当的时候重新执行逆向操作。由于并发操作过程的特点，所以需要引入多线程概念并使用线程池控制线程规模，以下首先给出相关的线程实现代码：

```
.....
// 这是支持并发调用的正向执行调用
private class ParallelCallable implements Callable<Response> {
    private RemoteInfo remoteInfo;
    // 上下文信息：由于各原子服务都是无状态的，所以需要上下文标示进行业务状态记录
    private CommandContext commandContext;
```



```

        private Request request;
        public ParallelCallable(RemoteInfo remoteInfo , CommandContext
commandContext , Request request) {
            .....
        }
        @Override
        public Response call() throws Exception {
            Response response = null;
            try {
                ParallelInvoker.this.openRemoteConnect(this.remoteInfo);
                ExecuteCommand.Client client = (ExecuteCommand.Client)
remoteInfo.getExecuteClient();
                response = client.execute(request, commandContext);
                if(response == null) {
                    response = new Response();
                    response.setResponseCode(RESCODE.ERROR);
                    response.setResponseValue("");
                }
            } catch (TException e) {
                ParallelInvoker.LOGGER.error(e.getMessage() , e);
                response = new Response();
                response.setResponseCode(RESCODE.ERROR);
                response.setResponseValue(e.getMessage());
            } finally {
                //最后都要关闭连接
                .....
            }
            return response;
        }
    }
    // 这是支持并发调用的逆向执行操作
    private class ParallelReverseCallable implements Callable<Response> {
        private RemoteInfo remoteInfo;
        // 上下文信息: 由于各原子服务都是无状态的, 所以需要上下文标示进行业务状态记录
        private CommandContext commandContext;
        public ParallelReverseCallable(RemoteInfo remoteInfo , CommandContext
commandContext) {
            .....
        }
        @Override
        public Response call() throws Exception {
            Response response = new Response();
            try {
                ParallelInvoker.this.openRemoteConnect(this.remoteInfo);

```

```

        UndoCommand.Client client = (UndoCommand.Client)
remoteInfo.getUndoClient();
        client.undo(commandContext);
        response.setResponseCode(RESCODE.SUCCESS);
    } catch (TException e) {
        ParallelInvoker.LOGGER.error(e.getMessage(), e);
        response.setResponseCode(RESCODE.ERROR);
        response.setResponseValue(e.getMessage());
    } finally {
        //最后都要关闭连接
        .....
    }
    return response;
}
}
.....

```

以上两个线程分别定义了正向业务调用和对应的逆向业务调用，其中的处理非常简单，就是使用 Apache Thrift 的客户端进行服务器端相应的业务调用。以下是 Invoker 角色中主要的代码片段：

```

.....
@Test
public void execute() {
    /*
    * 执行过程为：
    * 1. 首先多线程调用必须使用线程池控制线程规模，
    * 这个原因在本书第 7 章 7.8 节中已经介绍
    * 2. 准备好测试的服务器端后，进行并发调用——分布式业务各原子服务的正向执行
    * 3. 并发调用并等待所有原子服务都执行完成后，检测所有的原子服务是否都运行成功了
    * 4. 如果执行成功了，则不再进行后续操作；如果没有成功，就要进行逆向操作
    * 5. 逆向操作也需要检查成功性，如果逆向操作不成功，
    * 则必须进行日志记录并在后续的合适时再次逆向操作
    *
    * 注意：实际上以上每个步骤无论成功与否都要进行日志记录
    */
    // 这个线程池用来并发执行各个远程服务的调用，这些服务间基本没有依赖性，各自可
    // 以独立工作
    ExecutorService executorService = Executors.newFixedThreadPool(5);
    // 这里建立 Command 上下文，目前里面只有一个唯一编号的 uid 信息
    ExecutorService executorService = Executors.newFixedThreadPool(5);
    // 如果是正式系统，那么里面的信息肯定还有很多
    CommandContext commandContext = new CommandContext();
    commandContext.setCommandUid(UUID.randomUUID().toString());
}

```



```

// 2. ===== (步骤1 没有实际代码)
boolean success = true;
List<Future<Response>> serviceFutrues = new ArrayList<Future
<Response>>(this.remoteList.size());
Request request = new Request();
request.setParamname1("paramname");
request.setParamValue("paramvalue");
for(int index = 0 ; index < this.remoteList.size() ; index++) {
    RemoteInfo remoteInfo = this.remoteList.get(index);
    ParallelCallable parallelCallable = new ParallelCallable
(remoteInfo , commandContext , request);

    // 结果信息放置在这里
    Future<Response> future = executorService.submit(parallelCallable);
    serviceFutrues.add(future);
}
// 3. =====检测各原子服务返回的结果有无异常
for (int index = 0 ; index < serviceFutrues.size() ; index++) {
    Future<Response> future = serviceFutrues.get(index);
    try {
        Response response = future.get();
        // 如果条件成立, 则表示调用失败
        if(response == null || response.responseCode == RESCODE.ERROR) {
            success = false;
            break;
        }
    } catch (InterruptedException | ExecutionException e) {
        // 如果有异常抛出, 也说明调用失败
        LOGGER.error(e.getMessage() , e);
        success = false;
        break;
    } finally {
        // TODO 无论执行是否成功, 在这里都应该记录日志
    }
}
// 如果条件成立, 那么说明分布式业务执行成功了, 后续的逆向操作过程就不需要执行了
if(success) { return; }
// 4. =====
// 如果以上条件不成立, 就说明分布式业务执行失败, 那么进行逆向操作
List<Future<Response>> reverseFutrues = new ArrayList<Future
<Response>>(this.remoteList.size());
for(int index = 0 ; index < this.remoteList.size() ; index++) {
    RemoteInfo remoteInfo = this.remoteList.get(index);
    ParallelReverseCallable reverseCallable = new

```

```

ParallelReverseCallable(remoteInfo , commandContext);
    // 结果信息放置在这里
    Future<Response> future = executorService.submit
(reverseCallable);
    reverseFutrues.add(future);
}
// 检测逆向操作是否成功
success = true;
for (int index = 0 ; index < reverseFutrues.size() ; index++) {
    Future<Response> future = reverseFutrues.get(index);
    try {
        Response response = future.get();
        // 如果条件成立,则表示调用失败
        if(response == null || response.responseCode == RESCODE.
ERROR) {
            success = false;
            break;
        }
    } catch (InterruptedException | ExecutionException e) {
        // 如果有异常抛出,也说明调用失败
        LOGGER.error(e.getMessage() , e);
        success = false;
        break;
    }
}
// 如果逆向操作失败,则必须记录日志,以便在适当的时候再进行逆向操作
if(success) { return; }
//5. 这里必须要记录日志,因为逆向执行一旦失败,并且重试无效,
//就代表分布式业务的事务补偿机制失效,那么依据这个日志调用器将在后续合适的时间
//重新调用原子服务的逆向操作
//日志不能仅仅记录在内存中,或者输出到控制台上,而是应该存储到系统的持久化层,
//原因后文会进行讨论
this.recordErrorLog(commandContext);
}
.....

```

为什么日志需要持久化保存,而不能只记录在调用器的内存中呢?这是因为调用器存在单点故障,当它异常中断并被重启后,记录在内存中的日志就会消失。如果出现这样的情况,那么调用器恢复工作后既无法找到之前正在执行却突然中断的分布式服务过程,也无法找到出现逆操作异常等待重试的那些分布式业务。

日志的持久化操作可以直接以文件方式保存到文件系统中,不过为了保证不出现文件写冲突,保证文件写入效率,在写文件时就需要单个线程独占文件,并且对数据进行批量的顺序写

入操作。具体的策略也有很多，可以在多个线程对同一文件进行操作时，首先使用 FileChannel 的 tryLock 方法或者 lock 方法对文件进行独占锁定后再行操作，或者使用队列将待操作的数据送入一个专用线程再进行文件写操作（图 8-18）。

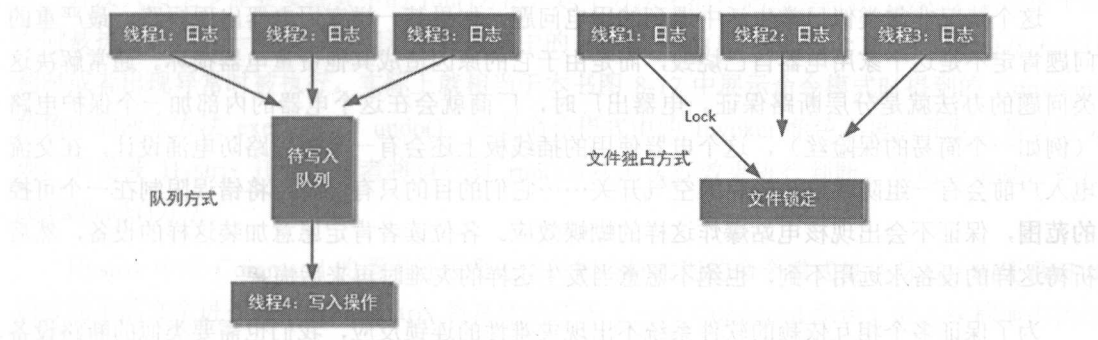


图 8-18 文件独占方式

如果使用现成的第三方存储系统，就不需要再过多考虑数据一致性的问题。不过本书并不建议使用关系型数据库作为日志数据的存储方案，因为存储吞吐量在高并发情况下不满足性能要求。而类似 LevelDB 这样的单节点单进程高吞吐量的组件就是一个不错的选择。

#### 4. Hystrix 与熔断

##### （1）Hystrix 是什么

一个独立运行的业务系统，完成不同业务所需要调用的各种第三方远程接口的总量上百，是再正常不过的事情了。无论是类似本节内容中描述的一个完整的分布式业务，还是单个完全独立被调用的远程服务接口，它们都可能存在工作异常的现象（只是分布式业务中的原子服务更明显，因为一旦有一个原子服务调用失败，那么整个分布式业务就会被视为调用失败）。试想一下，一个涉及 20 个原子服务的分布式业务，每一个原子服务的可用性都可以达到 99.999% 的稳定度，那么这个完整的分布式业务的稳定性就是 99.98%，也就是说一年也会有 105 分钟处于不可用的状态。如果换算成调用次数，就是一年 2 亿次完整的分布式业务过程会有 4 万次出现执行异常，需要进行分布式业务的逆向操作。

而一旦外部服务不可用，本系统最该担心的还并不是业务执行不成功，而是可能由此导致的连锁反应。例如业务系统有一个专门的线程池进行外部系统调用，但是由于外部系统 X 迟迟不给出反馈，造成本系统执行业务的线程在短时间内就达到最大阈值（max），并一直被占据。后续的业务调用请求就只有在 queue 中排队（队列是否报错，这要看所使用的 queue 的类型，这个在本书第 7.8 节也已经解释过），这不但使得本系统依赖系统 X 的所有业务无法执行，就连原本不依赖系统 X 的可以正常执行的其他业务也不能正常执行了。

这只是连锁反应的第一阶段，影响范围还可能继续扩散：由于本系统不再可用，导致依赖于本系统的更上层系统在短时间内也变得不可用。当连锁反应扩大到一定程度，就算这时最开始爆发错误的系统 X 恢复了工作，但是对整个顶层系统的负面影响也变得不可消除了。

这个情况非常类似日常生活中遇到的用电问题，如果某一样家用电器出现短路，最严重的问题肯定不是这个家用电器自己烧毁，而是由于它的原因造成其他贵重电器损坏。通常解决这类问题的办法就是分层断路保证。电器出厂时，厂商就会在这个电器的内部加一个保护电路（例如一个简易的保险丝），这个电器使用的插线板上还会有一个防短路防电涌设计，在交流电入户前会有一组防跳电防短路的空气开关……它们的目的只有一个：将错误限制在一个可控的范围，保证不会出现核电站爆炸这样的蝴蝶效应。各位读者肯定愿意加装这样的设备，然后祈祷这样的设备永远用不到，也绝不愿意当发生这样的灾难时再来后悔。

为了保证多个相互依赖的软件系统不出现灾难性的连锁反应，我们也需要类似的断路设备，在系统间调用出现异常时能将异常限制在可控范围，而 Hystrix 就起到了这样的作用。Hystrix 通过多种手段帮助技术人员解决系统间调用时的错误，防止系统雪崩效应。如果要使用 Hystrix，你需要在工程中引入相应的 Maven 库：

```
<dependency>
  <groupId>com.netflix.hystrix</groupId>
  <artifactId>hystrix-core</artifactId>
  <version>1.3.16</version>
</dependency>
<dependency>
  <groupId>com.netflix.hystrix</groupId>
  <artifactId>hystrix-metrics-event-stream</artifactId>
  <version>1.1.2</version>
</dependency>
```

要简单理解 Hystrix 的工作方式只需要理解几个关键点即可。首先 Hystrix 的单次执行是以一个命令完成的，其内是一个完整的命令模式，其外向开发人员暴露的是一个命令元素，就是本节内容（图 8-17）提到的命令模式中的 Command 角色。Hystrix 中的命令接口定义如下：

```
package com.netflix.hystrix;
.....
@ThreadSafe
public abstract class HystrixCommand<R> implements HystrixExecutable<R> {
  .....
  protected abstract R run() throws Exception;
  protected R getFallback() {
    throw new UnsupportedOperationException("No fallback
available.");
  }
}
```



```

.....
}
.....

```

开发人员至少需要实现其中的 `run()` 方法，这个方法中的代码内容将在 `Hystrix` 调用这个方法时被执行。但是一般来说我们还要重写其中的 `getFallback()` 方法，这个方法在 `run()` 方法运行超时或者出现异常时被执行。实际上就相当于本书图 8-17 中展示命令模式时提到的 `Command` 角色中的两个方法 `execute()` 和 `undo()`。只是命令模式中的 `Invoker` 角色不再由开发人员进行实现，而是被 `Hystrix` 托管，后者将自行对 `run()` 方法的执行效果进行判断，从而决定是否执行 `getFallback()` 方法。

`Hystrix` 中对 `Command` 的调用过程要远比我们自行实现的命令模式复杂得多，但还是可以通过简单的文字进行总结的：`Hystrix` 将具体的任务（一个 `Command` 角色）放入线程池中执行，而且 `Hystrix` 中不但可以管理一个线程池，还可以为不同的 `Command` 设置不同的 `ThreadPoolKey`，让它们工作在不同的线程池中。以下代码是 `Hystrix` 中创建线程池的默认过程，你可以在一个名叫 `HystrixConcurrencyStrategy` 的类中找到这些代码片段：

```

.....
public ThreadPoolExecutor getThreadPool(final HystrixThreadPoolKey
threadPoolKey, HystrixProperty<Integer> corePoolSize, HystrixProperty<Integer>
maximumPoolSize, HystrixProperty<Integer> keepAliveTime, TimeUnit unit,
BlockingQueue<Runnable> workQueue) {
    return new ThreadPoolExecutor(corePoolSize.get(), maximumPoolSize.get(),
keepAliveTime.get(), unit, workQueue, new ThreadFactory() {
        protected final AtomicInteger threadNumber = new AtomicInteger(0);
        @Override
        public Thread newThread(Runnable r) {
            // 线程池中的线程名称都带有 threadPoolKey 的名字
            return new Thread(r, "hystrix-" + threadPoolKey.name() + "-" +
threadNumber.incrementAndGet());
        }
    });
}
.....
public BlockingQueue<Runnable> getBlockingQueue(int maxQueueSize) {
    // 关于 SynchronousQueue 和 LinkedBlockingQueue 队列的特点
    // 可以参见本书第 7.8 节中的内容
    if (maxQueueSize <= 0) {
        return new SynchronousQueue<Runnable>();
    } else {
        return new LinkedBlockingQueue<Runnable>(maxQueueSize);
    }
}

```



```

}
.....

```

如果开发人员在创建 `Command` 时没有明确指定 `ThreadPoolKey`, `Hystrix` 也会使用 `CommandGroupKey` 作为线程池分组的依据。你还可以在 `HystrixThreadPool.Factory` 中观察到 `Hystrix` 对多个线程池的管理机制:

```

.....
public interface HystrixThreadPool {
    .....
    static class Factory {
        /*
         * Use the String from HystrixThreadPoolKey.name() instead
         * of the HystrixThreadPoolKey instance as it's just an
         * interface and we can't ensure the object
         * we receive implements hashCode/equals correctly and do
         * not want the default hashCode/equals which would create
         * a new threadpool for every object we get even if the
         * name is the same
         */
        private static ConcurrentHashMap<String, HystrixThreadPool>
threadPools = new ConcurrentHashMap<String, HystrixThreadPool>();
    }
    .....
}
.....

```

`Hystrix` 使用线程池进行 `Command` 的执行管理, 主要是为了控制一组 `Command` 的执行规模, 使得它们在远程系统出现响应延迟时, 后续的 `Command` 不会耗费宝贵的系统资源不断进行无用请求。一旦后续的 `Command` 无法进入线程池或者无法进入等待队列就视为 `Command` 执行失败, `Hystrix` 会直接执行这个 `Command` 对象的 `getFallback()` 方法。此外 `Hystrix` 还可以通过信号量 `Semaphore` 对一组 `Command` 的执行规模进行限制。

`Hystrix` 对 `Command` 的执行分为三种方式, 同步执行 `execute()`、异步执行 `queue()` 和带有观察器的执行方式 `observe()`, 这三种执行方法在 `HystrixExecutable` 接口中进行了详细定义:

```

package com.netflix.hystrix;
.....
/**
 * Common interface for executables ({@link HystrixCommand} and {@link
 * HystrixCollapser}) so client code can treat them the same and combine
 * in typed collections if desired.
 */
public interface HystrixExecutable<R> {

```

```

/**
 * Used for synchronous execution of command.
 * .....
 */
public R execute();
/**
 * Used for asynchronous execution of command
 * This will queue up the command on the thread pool and return an
 * {@link Future} to get the result once it completes
 * NOTE: If configured to not run in a separate thread, this will
 * have the same effect as {@link #execute()} and will block
 * We don't throw an exception in that case but just flip to
 * synchronous execution so code doesn't need to change in order to
 * switch a circuit from running a separate thread to the calling thread
 */
public Future<R> queue();
/**
 * Used for asynchronous execution of command with a callback by
 * subscribing to the {@link Observable}
 * This eagerly starts execution of the command the same as {@link
 * #queue()} and {@link #execute()}
 * A lazy {@link Observable} can be obtained from {@link HystrixCommand#
 * toObservable()} or {@link HystrixCommand#toObservable()}
 * <ul>
 * <li>When using {@link ExecutionIsolationStrategy#THREAD} this
 * defaults to using {@link Schedulers#threadPoolForComputation()}
 * for callbacks.</li>
 * <li>When using {@link ExecutionIsolationStrategy#SEMAPHORE} this
 * defaults to using {@link Schedulers#immediate()} for callbacks.</li>
 * </ul>
 * Use {@link HystrixCommand#toObservable(rx.Scheduler)} or {@link
 * HystrixCommand#toObservable(rx.Scheduler)} to schedule the
 * callback differently.
 */
public Observable<R> observe();
}

```

## (2) Hystrix 的基本使用

我们可以将业务系统中，一个不可拆分的粒度最小地对外部的服务调用，封装成 Hystrix 的一个 Command，例如“提交货运单”。并在 Command 定义的 `getFallback` 方法中（又被称为服务降级方法），为这个原子服务准备一个调用失败后的解决办法，例如记录一个本地日志，并将运单信息推送到远程的消息队列中。请看一下基于 Spring 工程定义的 Hystrix Command：

```

.....
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixCommandKey;
import com.netflix.hystrix.HystrixCommandProperties;
.....
// 使用 Hystrix 托管订单的服务调用, 由于一个 Command 只能被调用一次,
// 所以使用 Spring 时一定要使用 prototype 标记
@Component("HandleOrderInvoker")
@Scope("prototype")
public class HandleOrderInvoker extends HystrixCommand<Boolean> {
    private static final Log LOGGER = LogFactory.getLog(HandleOrderInvoker.
class);
    // 这是要提交的订单
    private Order order;
    // 这是真正的远程调用接口, 使用 RPC 完成
    @Autowired
    private OrderRemoteService orderRemoteService;
    public HandleOrderInvoker(Order order) {

        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("groupNam
e"))

        .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
            // 设置单次执行一个完整的分布式业务的超时时间, 设置 10 秒超时
            .withExecutionIsolationThreadTimeoutInMilliseconds(10000)
            .withExecutionIsolationThreadInterruptOnTimeout(true))
            // 这里最好明确设置 CommandKey 的信息

        .andCommandKey(HystrixCommandKey.Factory.asKey("HandleOrderInvoker")));
        this.order = order;
    }
    @Override
    protected Boolean run() throws Exception {
        try {
            // 此方法向远程业务系统提交订单信息
            this.orderRemoteService.sendOrder(this.order);
        } catch (Exception e) {
            LOGGER.error(e.getMessage(), e);

```



```

        throw e;
    }
    return true;
}
// 一旦正向执行过程出现异常，或者在规定的时间内没有完成执行，则执行服务降级逻辑
@Override
protected Boolean getFallback() {
    // 这里发送 order 信息到远端消息队列
    // 如果发送消息队列成功，则返回 true，否则返回 false
    return true;
}
}

```

在以上的代码中，虽然我们可以直接调用远端接口 `OrderRemoteService`，但是我们并没有这样做，而是使用了 `HystrixCommand` 对接口进行了封装，转由 `Hystrix` 帮我们执行这个远程调用。如图 8-19 所示为工程内部的层次结构转变。

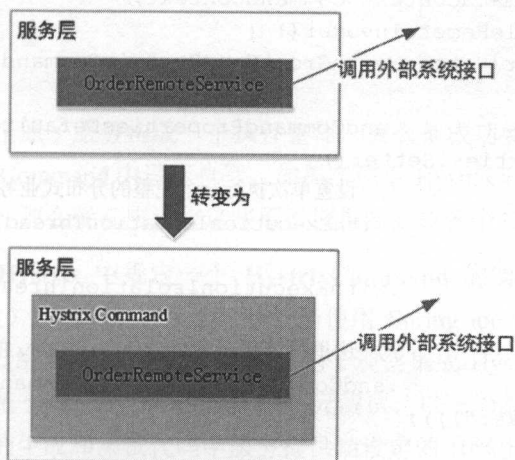


图 8-19 使用 `HystrixCommand` 封装外部调用

### (3) 对事务补偿实现进行调整

在之前事务补偿机制的示例中，无论是对分布式业务的顺序执行还是并发执行中，整个分布式业务是否需要逆向操作完全是技术人员通过远端业务服务调用的返回结果/异常自行判断，然后再决定是否需要进行逆操作，更没有错误熔断机制保证本系统其他业务执行不受影响——因为 `Feature` 的 `get()` 方法位置可能会由于各种原因长时间不返回执行结果。现在我们通过引入 `Hystrix`，将分布式业务的正向操作和逆向操作分离开，并通过设置最长等待时间、抛出异常等方式让逆向操作自动执行。以下代码片段演示了这个拆分过程。注意，由于我们只是对正向和

逆向操作的执行过程进行了拆分，但是涉及的执行逻辑却没有变化，所以为了节约篇幅，run()方法和 getFallback()方法中的重复代码就不进行展示了，读者可以通过下载示例工程对详细代码进行查看：

```

.....
@Component("HandleRemoteInvoker")
@Scope("prototype")
public class HandleRemoteInvoker extends HystrixCommand<String> {
    .....
    //这个集合一旦执行顺序确认后就不会有写操作了，
    //而在后续的执行过程中，这个集合中的数据索引位会被反复读取，所以选用 Array 形式
    private List<RemoteInfo> remoteList = new ArrayList<RemoteInfo>();
    // 远程业务并发调用所使用的线程池。也是使用 Spring 进行托管的
    @Autowired
    private ExecutorService executorService;
    //这里建立 Command 上下文，目前里面只有一个唯一编号的 uid 信息
    private CommandContext commandContext;
    public HandleRemoteInvoker() {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.
asKey("groupName"))
                                .andCommandPropertiesDefaults
(HystrixCommandProperties.Setter()
                                // 设置单次执行一个完整的分布式业务的超时时间，设置 1 秒超时
                                .withExecutionIsolationThreadTimeoutInMilliseconds
(1000)
                                .withExecutionIsolationThreadInterruptOnTimeout
(true))
                                // 这里最好明确设置 CommandKey 的信息
                                .andCommandKey(HystrixCommandKey.Factory.asKey
("ExecuteHystrixInvoker"))));
    }
    // 在这个位置写入需要调用的过程信息，测试的原子服务地址
    @PostConstruct
    public void init() {
        // 第一个原子业务
        RemoteInfo remoteInfo = new RemoteInfo();
        remoteInfo.setIp("127.0.0.1");
        remoteInfo.setPort(8081);
        remoteList.add(remoteInfo);
        // 第二个原子业务
        remoteInfo = new RemoteInfo();
        remoteInfo.setIp("127.0.0.1");
        remoteInfo.setPort(8082);
        remoteList.add(remoteInfo);
    }
}

```



```

        this.commandContext = new CommandContext();
        this.commandContext.setCommandUid(UUID.randomUUID());
toString());
    }
    @Override
    protected String run() throws Exception {
        //进行正向操作的过程已经在{@ ParallelInvoker}和{@ SerialInvoker}
        //中说明了
        //这里就不再进行赘述了
        .....
    }
    // 一旦正向执行过程出现异常，或者在规定的时间内没有完成执行，则自动进行逆向执行
    @Override
    protected String getFallback() {
        //进行逆向操作的过程也已经在{@ ParallelInvoker}和{@ SerialInvoker}中说明了
        //读者可以下载完整工程查看这里就不再进行赘述了
        .....
    }
    .....
}

```

由于在本示例中多个原子服务构成一个执行整体，要么都成功要么都逆向操作，所以我们将这个过程封装到一个 `Command` 中进行执行。如果在使用 `Hystrix` 时，你的各个原子调用不需要构成相互依赖的关系，那么就都应该将这些原子调用封装成各个独立的 `Command`。

读者还需要注意，`Hystrix` 中规定一个 `HystrixCommand` 的实例只能被执行一次（即是 `execute` 方法被调用一次），否则就会报错。所以当使用 `Spring ioc` 容器进行 `HystrixCommand` 类托管时，一定要使用 `@Scope("prototype")` 标记。为了观察集成 `Hystrix` 后 `Invoker` 角色的工作效果，笔者又写了一个基于 `Spring Boot Test` 的单元测试，以下为部分测试代码和可能出现的测试效果（因为 `UUID` 的生成和测试代码中服务提供端设定的 10% 出错概率都是随机的）：

```

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SpringBootTest.class)
public class Invoker {
    //为什么这个测试里面要注入多次 HystrixCommand
    //主要是为了惯出 Spring ioc 容器在 @Scope 标注下对 HandleRemoteInvoker 的实
    //例化情况
    @Autowired
    private HystrixCommand<String> handleRemoteInvoker1;
    @Autowired
    private HystrixCommand<String> handleRemoteInvoker2;
    private static final Log LOGGER = LoggerFactory.getLog(Invoker.class);
    @Test

```

```

public void testInvoker() {
    String result = handleRemoteInvoker1.execute();
    LOGGER.info("result = " + result);
    result = handleRemoteInvoker2.execute();
    LOGGER.info("result = " + result);
}
}

```

以下是可能出现的执行效果:

```

.....
15:43:03.330 [pool-2-thread-2] ERROR org.apache.thrift.ProcessFunction -
Internal error processing execute
org.apache.thrift.TException: FirstRemoteCommand execute exception !
    at test.command.server.FirstExecuteCommand.execute (FirstExecuteCommand.
java:51)
        at test.command.rpc.ExecuteCommand$Processor$execute.getResult
(ExecuteCommand.java:174)
            at test.command.rpc.ExecuteCommand$Processor$execute.getResult
(ExecuteCommand.java:1)
                at org.apache.thrift.ProcessFunction.process(ProcessFunction.
java:39)
                    at org.apache.thrift.TBaseProcessor.process(TBaseProcessor.java:39)
                        at org.apache.thrift.TMultiplexedProcessor.process(TMultiplexedProcessor.
java:123)
                            at org.apache.thrift.server.TThreadPoolServer$WorkerProcess.run
(TThreadPoolServer.java:286)
                                at java.util.concurrent.ThreadPoolExecutor.runWorker
(ThreadPoolExecutor.java:1142)
                                    at java.util.concurrent.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.
java:617)
  at java.lang.Thread.run(Thread.java:745)
15:43:03.330 [pool-1-thread-2] INFO test.command.server.SecondExecuteCommand
- SecondRemoteCommand execute , command uid : c8256812-34ee-4085-bc58-
bacb56c45b2c , response code is ERROR
15:43:03.347 [pool-2-thread-3] INFO test.command.server.FirstUndoCommand
- FirstRemoteCommand command uid : c8256812-34ee-4085-bc58-bacb56c45b2c now
undo!
15:43:03.348 [pool-1-thread-3] INFO test.command.server.SecondUndoCommand -
SecondRemoteCommand command uid : c8256812-34ee-4085-bc58-bacb56c45b2c now
undo!
.....

```

# 调用者返回的日志信息如下:

```
2017-02-07 15:43:25.004 INFO 18125 --- [main]
test.command.hystrixInvoker.Invoker: result = FALLBACK
```

#### (4) Hystrix 非常强大

本部分只是展示了 Hystrix 最简单的使用方式，目的是告诉读者事务补偿机制的理论有许多实现手段，并且可以非常深入地进行优化。Hystrix 除了能够帮助我们封装外部的原子服务完成调用过程，并在调用出现异常的情况下继续保持服务的健壮性，还提供了诸如服务限流、并发 RPC 调用等功能。

Hystrix 功能非常强大，以至于在 Spring Cloud 服务治理框架中直接对 Hystrix 进行了集成，开发人员只需要通过非常简单的注解就可以在自己的 Spring Cloud 工程中使用它，用于隔离和处理系统间调用过程中出现的异常情况。如果你的项目中没有使用 Spring Cloud 也没有关系，由上面所示的小例子可以看出 Hystrix 本来就可以和 Spring/Spring Boot 进行很好地集成。

Hystrix 官网上还有非常详细的介绍，各位读者可以参考：<https://github.com/Netflix/Hystrix/wiki>。查看适用 Hystrix 解决问题的各种场景，可参见：<https://github.com/Netflix/Hystrix/wiki/How-To-Use>。

#### 5. 其他说明

本小节介绍了解决分布式业务异常情况的几种方式，并分别列举了一种使用命令模式实现事务补偿机制的示例和一种集成使用 Hystrix 进行原子服务托管的事务补偿机制示例。虽然这两个示例如果要应用到实际工作环境中都还有相当的执行路径需要覆盖（例如第一个基于命令模式实现的事务补偿机制中，只有逆向操作出现异常时才进行了日志记录，而正向操作是否成功都没有记录日志），但是这两个示例需要达到的“为读者展示事务补偿机制如何工作”的目标却达到了。由于本章篇幅所限，这两个示例的代码都没有粘贴完整，为了便于读者查看完整的代码，本书将所涉及的示例工程上传到：<http://download.csdn.net/detail/yinwenjie/9750136>，供有兴趣的读者进行查阅和讨论。注意，这个工程引入了 SpringBoot 并使用 SpringBootTest 组件管理 JUnit 单元测试。Server 包中有一个 APP 类，它采用普通的进程启动方式初始化两个独立运行的 Apache Thrift Server 用于模拟两个相对独立的业务服务。如果读者已经知道 SpringBoot 的配置方式，并且有空闲的时间，可以将这个 APP 类修改成使用 SpringBoot 方式启动（图 8-20）。

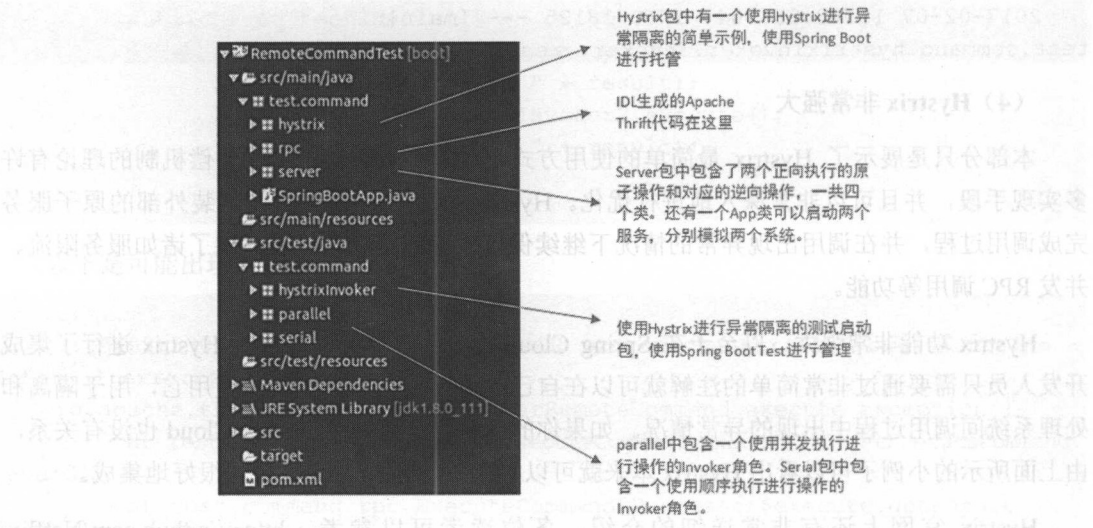


图 8-20 示例工程结构

## 8.5 SOA 和服务治理

系统间服务调用的过程远比本章内容中列举的场景要复杂得多, 需要考虑的问题也得多。试想一个单一的大型系统, 对外提供了 100 多个可独立工作的业务服务接口, 而它自己也需要调用的多个其他系统的接口, 加起来也不少于 100 个, 那么管理这些业务服务接口就需要耗费巨大的工作量。系统间服务调用的复杂性, 还体现在以下方面。

- 访问权限问题

在整个系统生态环境中, 不是任何用户都可以随意访问任意业务服务接口的。除了访问接口的用户组、用户和密码管理 (或者是公私钥文件), 还需要限制用户的访问权限。例如规定只有用户组 A 下所有的用户才能够访问接口 X。

- 版本控制问题

在比较大型的系统中, 某一个子系统一般都是使用集群环境运行 (有多个工作节点), 且必须 24 小时连续工作不允许中断。要进行某个服务接口升级, 怎么办? 一般的处理方式是, 先升级一部分服务节点, 其间让另一部分服务节点继续提供服务。这就可能造成在一个时间段同一个子系统提供的某一个服务出现服务接口版本的不兼容。为了子系统能够保证 24 小时连续提供服务, 就需要标注服务接口的版本号, 让之前没有完成升级的服务节点提供/访问老版本的服务接口; 已经完成升级的服务节点提供新版本的服务接口。



- 服务时效、次数控制问题

各个子系统提供的服务本身也是具有时效性的。比如某一个服务 Y 只在每天早上 10:00~11:00 才能提供访问调用,且对于某个用户来说每天只能调用 100 次。再比如说,某个接口 X 向 A 系统开放每天 100 万次的调用数量,但是向 B 系统只开放每天 10 万次的调用数量。

- 性能措施问题

系统服务接口能够承受多大的 TPS 是衡量其性能的一个重要指标。但是在生产环境下,往往再高的系统接口都会遇到 TPS 瓶颈。在此种情况下,我们一般会准备备用手段对请求进行导流,例如当接口 Y 的 TPS 接近 20000 时,将该接口 30% 的请求导流到接口 X,而接口 X 的业务逻辑是对这些请求进行暂存,并在系统资源空闲时再进行业务过程操作。

- 跨平台性问题

这类问题常出现在有历史遗留情况的系统中,或者有多个技术背景不同的团队同时进行研发的大中型业务系统中,这就要求远程业务接口能够支持多种语言客户端的调用。目前对这类问题的解决方法无外乎几种:一种是使用各开发语言都更容易理解和管理的基于网络应用层协议的接口调用方式,例如 HTTP RESTful 形式的服务接口;一种是基于某种支持跨语言特性的 RPC 组件提供接口支持,例如基于 Apache Thrift 提供的服务接口;还有一种是代理装置,通过这个代理装置,将某种语言某种消息格式的调用转换为另一种语言另一种消息格式的调用,而 ESB 技术就类似于这样一种代理装置。

## 8.5.1 SOA 概述

既然多个系统间的服务调用存在这么多现实问题,那么引入对服务接口进行专业化管理的独立组件/软件就是一个比较迫切的问题了。实际上业界早有解决这类问题的一个较高层面的定义,就是面向服务的架构——SOA,只不过它只是解决问题的一种思考方式,而并非具体的解决思路、解决办法。

SOA (Service-Oriented Architecture) 中文全称“面向服务的架构”。放在当下的技术环境(2015/2016 年),SOA 并不是一个新的概念。但是在 SOA 刚流行起来的 2000 年初(SOA 的概念最初由 Gartner Group 在 1996 年提出),这个架构模型就已经非常新颖了。本小节笔者试图用最平实的语言向各位读者介绍 SOA 概念中的几个核心内容。

SOA 主要围绕多个“服务”如何进行集成以达到某种服务目的进行讨论。那么 SOA 中所定义的服务是什么意思呢?在业务系统中被发布出来供用户使用,并能够完成一个完整业务过程的功能,就是服务(图 8-21)。



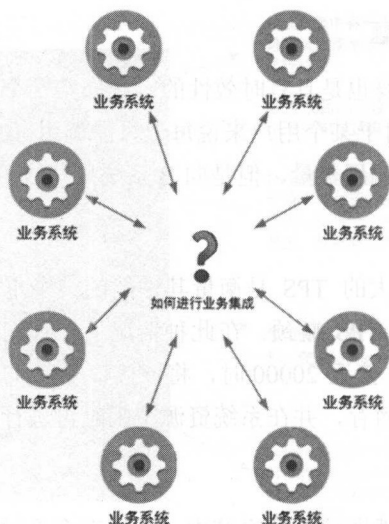


图 8-21 各子系统如何组织

- 服务着眼于完整的业务

从以上对“服务”的定义可以看出，服务的定义对象是业务系统中的完整业务功能。例如电商系统中“确认订单”这个功能就是一个服务，计费系统中“当月费用结算”功能就是一个服务；但是 CRM 系统中，需要完成“工单生成”功能所进行的“用户登录”动作就不是 CRM 系统中的服务定义，因为使用者进行“用户登录”是为了完成“工单生成”功能的权限验证步骤，并不是为了“登录”而登录。

- 服务的粒度虽然相对粗放，但却是可控的；服务拆分的目标是保留重用度

接下来讨论一下，“用户登录”这个功能在哪些情况下可以满足“服务”的定义呢？当用户中心系统为其他业务系统提供的“用户登录”功能被公布出来时这个“用户登录”可以被称为服务。服务粒度的拆分完全依据业务系统中业务过程进行定义和分析，所以服务的粒度都相对粗放。就像上一段文字中所举例的“费用结算”服务那样：可能完成“费用结算”功能，在计费系统内部需要完成“用户身份确认→上月费用查询→套餐清单查询→费用明细生成”这几个过程。但是这些过程都不会有任何其他业务系统进行单独使用，也就是说单独公布这些处理细节对于其他业务系统来说没有任何意义。

如果在后续的业务变更/业务重编时，业务设计人员发现某一个业务系统需要单独使用计费系统的“上月费用查询”功能，这时就需要将“计费系统”这个功能作为一个独立的服务向第三方系统公布出来。此时，这个功能就满足了“服务”的定义。

- 服务集成的目的是形成一个新的服务

对企业内部（或者企业间）的业务服务进行集成，被集成的业务服务称之为原子服务，集成的目的是重用这些原子服务形成一个新的服务。这样保证了技术团队/业务团队能以最小的代价，最高的效率使用既有服务，但同时也对实现 SOA 架构思想的软件提出了更高的要求。

- SOA 需保证屏蔽细节

使用 SOA 架构思想构建多个业务系统的集成关系，需要保证每个业务系统屏蔽细节。这些细节包括技术细节和业务细节。从技术细节层面看，无论业务系统使用哪种开发语言、哪种对外传输协议、哪种消息格式都可以使用 SOA 进行集成，并且能够在 SOA 架构的实现软件上完成不同传输协议的转换和不同消息格式的转换；从业务细节层面看，SOA 需要屏蔽业务系统的功能步骤细节。也就是说第三方系统只需要知道调用某一个服务就可以达到业务目的，至于提供服务的业务系统如何实现业务过程则无须关心。

- SOA 让各业务系统保持松散

SOA 架构模型为多个业务系统进行松散集成提供了一个良好的思路：通过屏蔽各业务系统技术细节和业务细节，兼容各业务系统的不同传输协议和不同消息格式，可以让通过 SOA 进行业务集成的，各个业务系统保持低耦合状态。这是因为所有协议和消息格式都处于开放状态，业务集成时，各业务系统不需要单独进行额外的转换工作，甚至不需要为基于 SOA 的业务集成进行任何额外工作，有时甚至无须知道对方系统的存在。

## 8.5.2 ESB 概述

既然 SOA 架构模型是解决问题的纲领性思路，那么在这个规则下有没有一些可行的具体解决办法呢？当然是有的，而且经过许多年的实际应用，这个解决办法已经非常成熟了，它就是 ESB（Enterprise Service Bus，企业服务总线）。既然是一种具体办法，就有这一具体办法的侧重考虑点，以及需要解决问题的适用环境。

企业的信息化建设一般要经历很长时间的发展，少则五六年多则十几年。所以我们看到某大型企业的信息系统最可能的情况是：存在着多个业务系统，甚至各业务系统负责的功能职责还存在重叠。这些系统采用不同时代的编程语言、编程框架、通信协议、消息格式和存储方案。

例如，计费系统可能采用 C++ 进行编写，对外调用功能采用 CORBA；年代久远的 CRM 系统采用 Delphi 进行编写，同样使用 CORBA 对外发布调用，并且最近两年该企业刚对 CRM 系统使用 C# 语言进行了一次升级，但是由于数据存储层的设计原因，并没有将老系统的所有数据割接到新系统，所以目前两套 CRM 系统都在使用；最新开发的财务联动系统，采用 Java 语言开发，并且不再采用应用程序窗口，改为使用浏览器进行页面展示和用户操作。这个财务联动系统多数对外的服务采用 HTTP 协议公布，还有一部分服务采用 Thrift RPC 对外公布……

由此可见, 由于各种可见的和不可见的原因, 企业信息化系统的建设历史和现实存在往往纷繁复杂。如果这些系统需要进行服务集成, 但是又没有一个成熟稳定、兼容易用的中间层进行协调, 那么要达到以上的调用要求基本上是不可能的 (即使实现也相当难以维护和扩展)。

为了满足 SOA 架构思想的设计要点, 达到既定的工作目标, ESB 总线技术至少需要帮助这些业务系统完成以下工作。

- 多种调用协议的兼容支撑和转换

无论业务系统向外部公布的服务使用哪种调用协议, 都可以通过 ESB 技术进行兼容性转换。例如 A 业务系统的服务只接受 Web Service SOAP 形式的调用, B 业务系统的服务却可以使用 Thrift RPC 进行调用 (不必为了调用 A 业务系统而专门去适应 A 业务系统的协议)。再基于 ESB 服务的中间层帮助实现两种协议的转换。

- 多种消息格式兼容支撑和转换

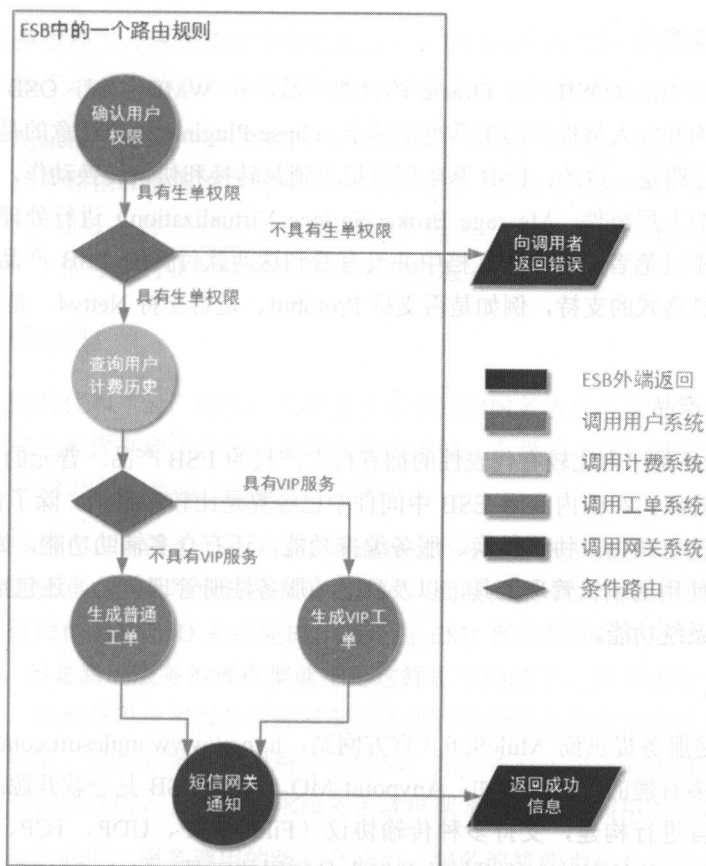
无论调用协议携带哪一种消息描述格式, 通过 ESB 中间件都可以实现相互转换。ESB 中间件应该支持将 JSON 格式的信息描述转换成目标业务系统能够识别的 XML 格式, 或者将 XML 描述格式转换成纯文本格式, 又或者实现两种不同结构的 XML 格式的互相转换, 等等。

- 服务监控管理 (注册、安全、版本、优先级)

既然 ESB 中间件要对原子服务进行集成, 考虑的问题就比较多了。首先, 业务系统提供的服务可能会以一定周期发生变化, 例如周期性的升级; 失控的业务系统甚至可能呈现完全无预兆无规律的服务变化, 例如突发性数据割接导致服务接口变动。那么 ESB 的实现软件中应该有一套功能, 能够保证在这样的情况下集成服务依然可以工作。其次, 并不是业务系统所提供的所有服务都可以在 ESB 中进行集成, 也并不是所有的服务都能被任何路由规则所编排, ESB 应该有一套完整的功能来保证服务集成的安全性和权限。

- 服务集成和编排

为了将多个服务通过 ESB 技术进行集成形成一个新的服务, ESB 技术必须能够进行服务编排。服务编排的作用就是明确原子服务执行的先后顺序、判断原子服务执行的条件、确保集成后的新服务能够按照业务设计者的要求正常工作。图 8-22 示例了新服务“工单派发”通过多个业务系统提供的原子服务, 按照设置的执行条件在 ESB 总线上进行工作的过程。



### 8.5.3 常见的 ESB 产品

常见的 ESB 产品主要分为商（付）用（费）产品和社区开（免）源（费）产品两类，由于篇幅原因，本书只对这些产品进行概述，感兴趣的读者可通过丰富的网络资源查看详细介绍。

#### • IBM ESB 系列产品

IBM 提供了两款纯软件的 ESB 产品：IBM WebSphere ESB 和 IBM WebSphere Message Broker (WMB)。后者是前者的升级版本（或者说是高级版本），支持更广泛的传输协议和消息格式。WebSphere Message Broker 产品是一个全分布式的 ESB 中间件产品，它使用多个 Message Broker 实例分别处理不同的业务编排、消息转换工作。并且使用 Message Broker Explorer 和 Message Broker Toolkit（基于 eclipse-plugin 的开发者工具）对这些 Broker 进行统一管理。

### • Oracle ESB 产品

Oracle Service Bus (OSB) 是 Oracle 的付费产品, 和 WMB 类似, OSB 也是全分布式的 ESB 产品, 并且为开发人员提供的工具也是基于 eclipse-Plugin。值得注意的是, OSB 和 WMB 实现业务编排的思路是一致的: ESB 基础层只提供消息转换和协议转换动作, 而服务编排和路由规则则由特定的上层组件 (Message Broker/Service Virtualization) 进行处理, 并且这些组件可以灵活部署。不过笔者从其官方文档中并没有看到这两款付费的 ESB 产品对一些目前流行的传输协议和消息格式的支持, 例如是否支持 Protobuf、是否支持 Netty4、是否支持 Avro, 等等。

### • 普元 ESB 产品

普元 ESB 产品是国内比较有代表性的拥有自主知识产权的 ESB 产品。普元的 ESB 中间件系统经过多年建设和升级, 在国内商用 ESB 中间件中已经算是比较完整了。除了包括 ESB 技术必要的消息转换、消息路由、协议转换、服务编排功能, 还有众多辅助功能, 如 ESB Studio 中提供给开发人员使用的编辑管理工具, 以及独立的服务注册管理单元, 还包括监控 ESB 中间件监控平台等子系统功能。

### • Mule ESB

Mule ESB 是服务提供商 MuleSoft (官方网站: <https://www.mulesoft.com/>) 的产品之一。这家公司还有很多有趣的产品, 例如: Anypoint MQ。Mule ESB 是一款开源的 ESB 产品, 主要基于 Java 语言进行构建, 支持多种传输协议 (File、FTP、UDP、TCP、Email、HTTP、SOAP、JMS 等), 并且有独立的提供给开发人员使用的工具: Mule Studio。消息格式转换方面, Mule ESB 提供了很多现成的组件, 使开发人员能在编排的服务内部轻松实现消息转换, 当然开发人员也可以按照 Mule ESB 的规范自行定义消息格式转换。

企业集成模式 (Enterprise Integration Patterns, 实际上是一本书名), 其中专门讨论的就是企业内部的各个业务系统如何进行集成。Mule ESB 中的业务服务集成就是基于 EIP 思想。

### • Apache Camel

Apache Camel 是 Apache 基金会下的一个顶级开源项目, 它提供了一套消息路由规则和消息转换引擎。不过它并没有现成的消息转换组件, 技术人员需要遵循 Camel 的定义规则自行实现消息转换 (你可以实现一次消息转换然后在各个路由定义中重复使用)。

Apache Camel 支持领域特定语言 (DSL), 所谓 DSL 是什么呢? 举个例子: 虽然物流行业的业务人员不懂编程, 但是他们理解物流行业的业务过程, 如果我们为他们提供一种特定的编程语言, 其中内置了所需物流行业的专有名词和业务过程, 这样业务人员就可以通过简单的



语法自行定义业务过程了。这就是领域特定语言——专门用在特定行业或领域进行业务规则描述的语言。Apache Camel 支持业务人员使用 Java 语言在业务集成领域进行规则描述。

严格来说, Apache Camel 并不能算作 ESB 中间件服务, 不过开发人员可以通过它获得完整的协议转换、消息路由和服务编排等能力, 然后再自行对它集成版本控制、服务注册、运行监控等其他能力并通过某种方式让这些服务运行起来。也就是说, 我们可以使用 Apache Camel 自行开发一个符合我们业务要求的 ESB 中间件服务。

#### 8.5.4 服务治理框架

在移动互联网时代和“云”时代, 又诞生了许多新的服务治理框架, 它们的目标是实现组件的轻量化、模块化, 提高调用性能以及降低集成难度。在满足 SOA 概念的前提下, 这些服务治理框架又在 ESB 功能侧重点上重新做了调整。

ESB 企业总线的存在目的之一是满足企业建设过程中新系统和遗留系统的集成问题, 所以 ESB 中的一个功能侧重点就是协议转换和信息转发, 而且如果 D 系统要对 A、B、C 等系统进行调用, 其调用过程也不是在 D 系统完成, 而是由 ESB 系统帮助其完成调用, 再将调用结果返回给 D 系统。但是新的服务治理框架就不是这样思考问题了, 后者通常采用“服务注册”的思路进行实现。也就是说这些服务治理框架并不进行原子服务的真实调用, 而只记录原子服务的调用地址、权限、熔断方案、备用路径等信息, 并在各原子服务调用时监控实时压力。当 D 系统需要调用外围系统时, 首先会请求服务治理框架拿到真实的调用路径, 再自行进行调用。

这样的做法虽然降低了服务调用的跨平台能力、服务编排能力和对遗留系统的集成能力, 但是显著提高了各系统间的调用吞吐量以及各系统集成到服务治理框架上的难度, 非常适合互联网系统对于轻量化和快速部署的要求。而且对于现在的互联网应用, 也没有那么多核心遗留系统需要进行服务集成, 所以适当牺牲各接口的跨语言特性和集成特性是可以接受的。

##### 1. 阿里 DUBBO

DUBBO 是一个分布式服务框架, 致力于提供高性能和透明化的 RPC 远程服务调用方案, 曾经是阿里巴巴 SOA 服务化治理方案的核心框架, 每天为 2 000 多个服务提供 3 000 000 000 多次访问量支持, 并被广泛应用于阿里巴巴集团的各成员站点。

——摘自 DUBBO 官网 (<http://dubbo.io/>)

由此可见, DUBBO 的设计思考在阿里集团内部的尝试是成功的, DUBBO 基于 Java 语言并完全兼容对 Spring 的集成, 使用 RPC 协议完成调用过程。这样的规则使得 DUBBO 集成到各工程的过程变得非常简单, 调用性能也非常不错, 但前提是需要使用 Java 语言, 如果你的系统是由 JS 语系实现的 (例如 Node.JS), 那显然就不能进行集成了, 另外如果你之前的系统

使用 RESTful HTTP 形式向外部提供接口，那么也不能使用 DUBBO 进行集成，除非对工程提供的外部接口做相应的 RPC 改造。这就是牺牲跨语言特性和协议转换能力，转而侧重于提高调用性能的一个典型实例。另外，这也是由于阿里内部各系统状态决定的——阿里集团内部并没有七八年都没有团队维护但还在线上作为核心系统运行的所谓“遗留系统”，业务系统在经历了必要的生命周期后就会在适当的时候被新的系统替换掉。

为了让部分还没有使用过 DUBBO 的读者对它有一个直观的认识，这里本书再花一些篇幅介绍一个简单的使用实例，对 DUBBO 更深入的分析可以参考笔者的博客内容：<http://blog.csdn.net/yinwenjie/article/details/50193987>。

### (1) 准备过程

- 你可以在以下站点下载后文中提到的各种组件：

DUBBO 官网和用户手册：<http://dubbo.io/>，<http://dubbo.io/User+Guide-zh.htm>。

DUBBO 服务治理框架 V2.4.10：<http://repo1.maven.org/maven2/com/alibaba/dubbo/2.4.10/>。  
这个版本是本书定稿时的版本号。

- 请在工程中引入 maven 的支持，并且导入相关依赖：

```
.....
<!-- dubbo -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>dubbo</artifactId>
    <version>2.4.10</version>
</dependency>
.....
```

请注意如果要使用 ZooKeeper 作为 DUBBO 服务管理仓库，那么还应该安装 ZooKeeper 服务器（测试环境下，单点服务即可）。

### (2) 定义和启动服务器端（Spring-XML 配置方式）

```
.....
<!-- 注解包扫描位置： -->
<context:component-scan base-package="yinwenjie.test.dubboService.*" />
<!-- 接入 DUBBO 的应用程序名称 -->
<dubbo:application name="ws-demo" />
<!-- 注册仓库地址：ZooKeeper 组件，192.168.61.128:2181 -->
<dubbo:registry address="zookeeper://192.168.61.128:2181" />
<!-- 用 dubbo 协议在 20880 端口暴露服务 -->
<dubbo:protocol name="dubbo" port="20880" />
<!--
```

声明需要暴露的服务接口，  
 请注意 ref 属性中指定的 MyService 接口实现类，它并没有在 xml 文件中定义，而是使用注解的方式在 class 中定义

```
-->
<dubbo:service    interface="yinwenjie.test.dubboService.iface.MyService"
ref="MyServiceImpl"/>
.....
```

### (3) 编写服务接口和实现

下面是 RPC 服务接口和服务接口实现的代码：

```
package yinwenjie.test.dubboService.iface;
//这是服务接口
public interface MyService {
    //这是一个测试接口
    //@param field1 参数 1
    //@param field2 参数 2
    // @return 相加后返回
    public String doMyTest(String field1 , String field2);
}
```

服务接口的实现：

```
package yinwenjie.test.dubboService.impl;
.....
@Component("MyServiceImpl")
public class MyServiceImpl implements MyService {
    @Override
    public String doMyTest(String field1, String field2) {

        return field1 + field2;
    }
}
.....
```

服务接口和服务接口的实现都是非常简单的 Java 代码，这里就不进行特别的说明了。只补充两个注意事项：

可以看出服务接口和服务接口的实现本身并没有引入和 RPC 结束任何相关的类，也就是说服务接口和服务接口的实现甚至自己都不知道将被某个 RPC 框架所调用。

dubbo: protocol 标签中，我们指定的协议名称为 dubbo（小写）。这里的 dubbo 是指 DUBBO 服务治理框架中自带的一种 RPC 协议。从官网给出的资料来看，DUBBO 服务治理框架还支持 rmi、hessian、HTTP、webService、Hessian 等多种通信协议。

以下代码片段可以启动集成了 DUBBO 的服务提供方：

```
package yinwenjie.test.dubboService;
.....
public class ServiceMainProcessor {
    .....
    public static void main(String[] args) throws Exception {
        new ClassPathXmlApplicationContext(new String[]{"application-service.xml"});
        //这里的同步锁和 DUBBO 框架本身的工作原理没有任何关系，只是为了让其不退出
        //当然你也可以使用 ClassPathXmlApplicationContext 中的 start 方法，效果一样
        synchronized (ServiceMainProcessor.WAITOBJECT) {
            ServiceMainProcessor.WAITOBJECT.wait();
        }
    }
}
.....
```

#### (4) 定义和启动客户端

以下是 RPC 服务消费方（RPC 客户端）的 XML 配置：

```
.....
<!-- 客户端应用程称呼名称 -->
<dubbo:application name="ws-demo-client" />
<!-- 注册仓库地址: zookeeper 组件, 192.168.61.128:2181 -->
<dubbo:registry address="zookeeper://192.168.61.128:2181" />
<!-- 引用的服务, 那个 interface 一定是一个被引入到 DUBBO 仓库的接口定义 -->
<dubbo:reference id="myService" interface="yinwenjie.test.dubboService.
iface.MyService"/>
.....
```

对于 DUBBO-RPC 框架来说，它其中的服务消费者并不需要进行多余的任何定义。只需要知道要访问的服务接口（和定义的服务版本，如果没有定义服务版本默认为 0.0.0）。下面的代码将启动 DUBBO-RPC 客户端，并且完成 RPC 调用：

```
.....
public static void main(String[] args) throws Exception {
    ApplicationContext app = new ClassPathXmlApplicationContext(new
String[]{"application-client.xml"});
    // 开始 RPC 调用
    MyService myService = (MyService)app.getBean("myService");
    LOGGER.info("myService = " + myService.doMyTest("1234", "abcde"));
    synchronized (ClientMainProcessor.WAITOBJECT) {
        ClientMainProcessor.WAITOBJECT.wait();
    }
}
```



## 2. Spring Cloud

由 Spring Source 主导开发的一款组件集合统称为 Spring Cloud，相对于 DUBBO 这种单纯为了满足服务治理轻量化而开发的组件而言，Spring Cloud 就要“重”很多了。以至于服务能力只是 Spring Cloud 涉及的其中一块（当然是很重要的也是最常被使用的一块），例如 Spring Cloud Config 可以为你的开发工程提供分布式配置信息管理功能，然后我们还可以配合 Spring Cloud Bus 将配置变化信息传递到每个服务节点。表 8-1 来源于官网，其中列举了 Spring Cloud 家族中主要的成员。

表 8-1

组件名	作 用
Spring Cloud Starters	Spring Boot-style starter projects to ease dependency management for consumers of Spring Cloud. (Discontinued as a project and merged with the other projects after Angel.SR2.)
Spring Cloud Config	Centralized external configuration management backed by a git repository. The configuration resources map directly to Spring 'Environment' but could be used by non-Spring applications if desired
Spring Cloud Netflix	Integration with various Netflix OSS components (Eureka, Hystrix, Zuul, Archaius, etc.)
Spring Cloud Bus	An event bus for linking services and service instances together with distributed messaging. Useful for propagating state changes across a cluster (e.g. config change events)
Spring Cloud Cluster	Leadership election and common stateful patterns with an abstraction and implementation for Zookeeper, Redis, Hazelcast, Consul
Spring Cloud Security	Provides support for load-balanced OAuth2 rest client and authentication header relays in a Zuul proxy
Spring Cloud Sleuth	Distributed tracing for Spring Cloud applications, compatible with Zipkin, HTrace and log-based (e.g. ELK) tracing
Spring Cloud Data Flow	A cloud-native orchestration service for composable microservice applications on modern runtimes. Easy-to-use DSL, drag-and-drop GUI, and REST-APIs together simplifies the overall orchestration of microservice based data pipelines



续表

组件名	作 用
Spring Cloud Stream	A lightweight event-driven microservices framework to quickly build applications that can connect to external systems. Simple declarative model to send and receive messages using Apache Kafka or RabbitMQ between Spring Boot apps
Spring Cloud Task	A short-lived microservices framework to quickly build applications that perform finite amounts of data processing. Simple declarative for adding both functional and non-functional features to Spring Boot apps
Spring Cloud Zookeeper	Service discovery and configuration management with Apache Zookeeper
Spring Cloud AWS	Easy integration with hosted Amazon Web Services. It offers a convenient way to interact with AWS provided services using well-known Spring idioms and APIs, such as the messaging or caching API. Developers can build their application around the hosted services without having to care about infrastructure or maintenance
Spring Cloud Connectors	Makes it easy for PaaS applications in a variety of platforms to connect to backend services like databases and message brokers (the project formerly known as "Spring Cloud")

这些组件大多数能够独立工作，当然我们会根据工程的实际情况将它们集成在一起使用。在实际开发过程中，我们最常使用的五个 Spring Cloud 组件分别是：服务发现管理组件（Spring Cloud Netflix Eureka）、服务路由/网管组件（Spring Cloud Netflix Zuul）、客户端负载组件（Spring Cloud Netflix Ribbon）、服务配置组件（Spring Cloud Config）以及断路器组件（Spring Cloud Netflix Hystrix）。请看，这些组件大多由 Netflix 提供，可见后者在 Spring Cloud 社区中举足轻重的作用，另外以上提到的常用组件中，除了客户端负载和服务配置组件，其他三个组件都和服务治理的概念相关。所以在有的资料中直接称呼 Spring Cloud 为微服务治理框架，而有的资料中则概括得更加广泛——微服务框架。

## 第 9 章

# 系统间通信：消息队列技术

从本章开始，本书介绍另一种类型的系统间通信及传输方案：消息队列。首先会讨论消息队列协议的基本原理和工作方式，并介绍几种目前生产环境中常用的消息协议：MQTT、XMPP、Stomp、AMQP、OpenWire 等。然后在这个基础上介绍一款消息队列产品：ActiveMQ，并通过一个虚拟的业务场景向读者展示如何将消息队列使用到业务环境中。本章的最后提到另一款消息队列产品，Apache Kafka，它也是现在业务系统中应用广泛的消息队列软件。我们还会讨论这些眼花缭乱的协议、软件、程序库之间的关系。

## 9.1 消息队列原理

### 9.1.1 消息

消息队列技术中（后文也简称 MQ）有一个基本概念需要在开篇前进行讨论：消息和消息协议。消息即是信息的载体，这个描述相信各位读者都能够明白。消息发送者需要知道如何构造消息；消息接收者需要知道如何解析消息。所以为了让消息发送者和消息接收者都能够明白消息所承载的信息，消息本身就需要按照一种统一的格式描述消息，这种统一的格式称之为消息协议。有效的消息一定具有某一种格式，而没有格式的消息是没有意义的。

消息从发送者到接收者的方式也有两种。一种我们可以称为直接消息通信，也就是说消息从一端发出后（消息发送者）直接发送消息给最终接收者，并得到处理后的回执——无论是阻塞方式还是非阻塞方式。这种通信方式的一种具体实现就是我们已经介绍过的 RPC；另一种方式可以称为中转消息通信，即消息从某一端发出后，首先会进入一个容器进行处理和临时存储，当消息接收者准备好或者达到某种传输条件后，再由这个容器发送给另一端，这种容器的一种具体实现就是消息队列。

9.1.2 服务结构

无论是 RPC 还是 MQ，它们的网络通信基础都建立在本书中已经介绍过的网络 I/O 模型之上（详见本书 7.1~7.3 节的内容）。先进的网络 I/O 模型将赋予 MQ 协议优异的性能表现，当然性能也不仅仅取决于网络 I/O 模型。

从图 9-1 可以看到，某一种消息通信组件（或者叫作程序库）的实现都建立在某种“消息协议”基础上，例如 gRPC 是一款 Google 推出的 RPC 组件，其主要使用 HTTP2 协议作为消息的描述格式，也支持原始的 HTTP 协议；RabbitMQ 消息通信组件属于一种 MQ 通信的实现，它主要支持 AMQP 消息格式协议，通过安装第三方插件还可以让它支持 MQTT 等消息协议；ActiveMQ 支持多种消息格式也属于一种 MQ 通信实现，例如 AMQP 协议、STOMP 协议和 MQTT 协议。

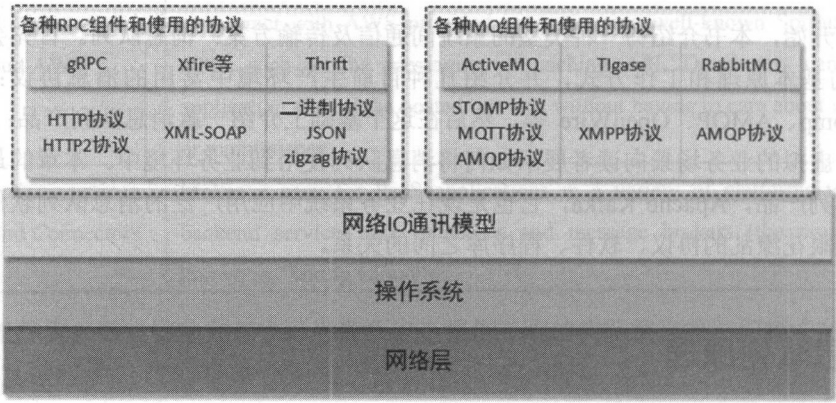


图 9-1 知识结构

虽然消息协议存在“私有协议”和“开放协议”之分（这主要看协议本身是否向行业开放规范文档，是否允许组织或者个人对协议进行实现），虽然某一个软件（程序库）不一定只支持一种协议，虽然某一种协议也不一定只由一种软件（程序库）实现，但是它们都适用消息通信组件都要基于一种或多种消息协议才能工作这个基本规则。

9.2 消息协议

要讨论 MQ，就应该从这些 MQ 支持的消息协议开始讨论。这个小节首先为各位读者介绍几种消息协议，它们是 XMPP、Stomp、MQTT 和 AMQP。

## 9.2.1 XMPP 协议

### 1. 基本定义

XMPP is the Extensible Messaging and Presence Protocol, a set of open technologies for instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data.

以上内容引用自 XMPP 官网，这个定义已经可以清楚地表明 XMPP 协议的用途和特性。XMPP 的前身是 Jabber，一个开源形式组织制定的网络即时通信协议（IM 通信）。XMPP 目前被 IETF 国际标准组织完成了标准化工作。

XMPP 基于 XML 进行 IM 系统的开发。国内比较流行的 XMPP 服务器叫作 Openfire，它使用 MINA 作为下层的网络 I/O 框架（不是 MINA2 而是 MINA1）；国外用得比较多的 XMPP 服务器叫作 Tigase，它的开发组织号称单个 Tigase 节点可以支撑 50 万用户在线，集群可以支持上亿用户同时在线（<http://projects.tigase.org/>）：

Cluster with over 1mn online users . 500k online users on a single machine

如果读者所在公司需要开发 IM 系统，那么除了使用现成的 XMPP 服务器，还需要实现了 XMPP 协议的客户端或者开发包（以便进行扩展开发）。你可以在 XMPP 官方网站上查看到 XMPP 官方推荐的开发包，各种语言的支持基本上都有：<http://xmpp.org/software/libraries.html>。

笔者曾参与过某几款 IM 系统的开发（包括自己创业的项目），总的来说，XMPP 协议本身是不错的选择，但是学习起来会耗费相当的时间，并且某些 XMPP 客户端、服务器端或者程序库并没有它的开发组织或者个人宣传得那么稳定好用。如果你的公司需要进行 IM 系统的开发，那么建立私有的消息协议或者另外寻找消息协议（例如 MQTT 协议）也会是一个不错的选择。

### 2. 协议通信过程示例

为了让各位读者对 XMPP 协议有一个初步认识，这里本书给出一个 XMPP 协议处理“IM 用户登录”操作的过程（XMPP 的登录方式分为有用户密码和无用户密码两种方式，这里介绍无密码登录方式）。XMPP 协议细节比较丰富，这里只讨论登录操作，如果读者有兴趣可以下载全套的 XMPP 官方规范文档进行研究（<http://xmpp.org/>）。

图 9-2 可以看到 XMPP 协议中的 xml 片段，这里出现了几个 XMPP 协议中的关键信息，如下所示。

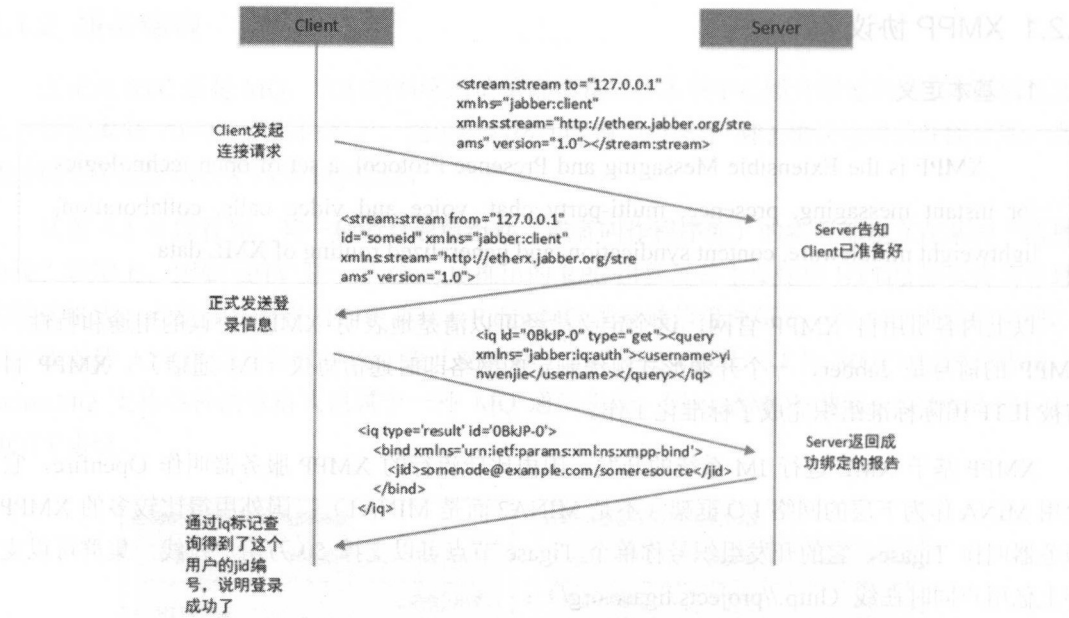


图 9-2 XMPP 通信过程实例

- stream 标记：通信流标记，是指 XMPP 的客户端或者服务器端向对方发起的通信请求（或者响应）。通信流并不携带真正的内容信息，只是表明客户端和服务端发生了一次交互。stream 的属性包括：to、from、id、xml:lang、version 等。
- iq 标记：iq 标记是 Info/Query 的简称（你可以理解成查询请求），一般是一对一的形式出现，由客户端发起查询请求，由服务器端返回查询结果。由于查询请求的类型不一样，iq 标记中可以嵌入的子标记就有很多。例如，可以嵌入 bind 标记，表明某个用户和 jid 的绑定关系；可以嵌入多个 item 标记，表明查询得到的这个用户的好友信息（如下）。

```
<iq to='somenode@example.com/someresource' type='result' id='roster'>
  <query xmlns='jabber:iq:roster'>
    <item jid='friend1@example.com' name='someone1'/>
    <item jid='friend2@example.com' name='someone2'/>
  </query>
</iq>
```

- jid 标记：jid（JabberID）是 XMPP 协议中的标识，它用来表示 XMPP 网络中的各个 XMPP 实体（实体可以是某一个用户、某一个服务器、某一个聊天室），规范格式如下：

```
jid = [ node "@" ] domain [ "/" resource ]
```

- 还有未出现的 message、presence 标记：message 是实体内容标记，记录了聊天的真实内



容；presence 标记表示了 XMPP 用户的服务状态（离线、在线、忙碌等）。示例如下：

```
<message to="somenode@example.com/someresource" type="chat">
  <body>helloworld.....</body>
</message>
```

## 9.2.2 Stomp 协议

### 1. 基本定义

Stomp 协议，英文全名 Streaming Text Orientated Message Protocol，中文名称为流文本定向消息协议。是一种以纯文本为载体的协议（以文本为载体的意思是它的消息格式规范中没有类似 XMPP 协议那样的 xml 格式要求，你可以将它看作“半结构化数据”）。目前 Stomp 协议有两个版本：V1.1 和 V1.2。

一个标准的 Stomp 协议包括以下部分：命令/信息关键字、头信息、文本内容。如图 9-3 所示。

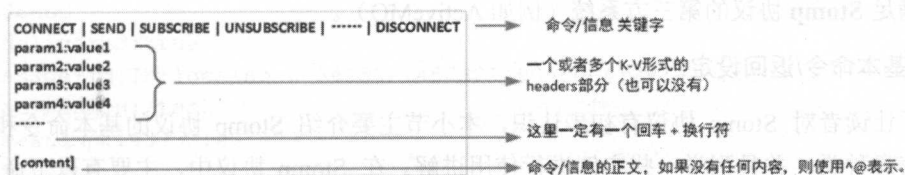


图 9-3 Stomp 协议格式

以下为一段简单的协议信息示例：

```
CONNECT
accept-version:1.2
someparam1:value1
someparam2:value2

this is conntecon ^@
```

上面的示例中，我们使用了 Stomp 协议的 `CONNECT` 命令，它的意思为连接到 Stomp 代理端，并且它携带了要求代理端的版本信息和两个自定义的 K-V 信息（请注意“^@”符号，Stomp 协议中用它来表示 NULL）。

Stomp 协议中有两个重要的角色：Stomp 客户端与任意 Stomp 消息代理（Broker）。如图 9-4 所示。

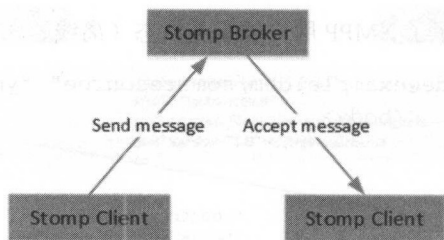


图 9-4 Stomp 消息转发示意

有的读者可能会问：为什么称为 Stomp 消息代理，而不称为 Stomp 消息服务？因为 Stomp Broker 只是负责接收和存储客户端发来的消息，只是按照客户端要求的路径转发消息，只是管理客户端连接和订阅：它并不负责根据消息内容做任何业务处理。所以将它称为消息代理端更贴切。

由于 Stomp 协议的结构如此简单，以至于任何理解 Stomp 协议命令格式的技术人员都可以开发 Stomp 的代理端或者 Stomp 的客户端，并将自己满足 Stomp 协议的系统轻松接入另一个同样满足 Stomp 协议的第三方系统（例如 ActiveMQ）。

## 2. 基本命令/返回设定

为了让读者对 Stomp 协议有初步认识，本小节主要介绍 Stomp 协议的基本命令和代理端返回的信息种类，并且列举一些实例进行使用讲解。在 Stomp 协议中，主要有以下命令/返回信息（有的文章中也称一个完整的信息为帧）。这些命令/返回信息构成了 Stomp 协议的主体，并能够支持 Stomp 客户端和 Stomp 代理端完成连接、发送、订阅、事务、响应等操作的整个过程。这些命令/返回如下。

- **CONNECT/STOMP 命令：**客户端通过使用 CONNECT 命令，连接到 Stomp 代理端。如果使用 STOMP 命令，那么 Stomp 代理端的版本必须是 1.2。
- **CONNECTED 信息：**当 Stomp 代理端收到客户端发送来的 Connect 命令并且处理成功后，将向这个客户端返回 CONNECTED 状态信息；如果这个过程中出现任何问题，还可能返回 ERROR 信息。
- **SEND 发送命令：**客户端使用 SEND 命令，向某个指定位置（代理端上的一个虚拟路径）发送内容。这样在这个路径上订阅了消息事件的其他客户端，将能够收到这个消息。
- **SUBSCRIBE 订阅命令：**客户端使用 SUBSCRIBE 订阅命令，向 Stomp 服务代理订阅某一个虚拟路径上的监听。这样当其他客户端使用 SEND 命令发送内容到这个路径上时，这个客户端就可以收到这个消息内容。在使用 SUBSCRIBE 时，有一个重要的 ACK 属性。这个 ACK 属性说明了 Stomp 服务代理端发送给这个客户端的消息是否需要等待收

到一个 ACK 命令，才认为这个消息处理成功了。如下所示：

```
SUBSCRIBE
id:XXXXXXXXX
destination:/test
ack:client

^@
```

以上 SUBSCRIBE 命令信息中，客户端订阅的虚拟位置是 test，且命令信息中 ACK 属性为 client。整个消息说明当客户端收到消息时，必须向代理端发送 ACK 命令，代理端才认为这个消息处理成功了（ACK 的值只有三种：auto（默认）、client 和 client-individual）。

- **UNSUBSCRIBE 退订命令：**客户端使用这个命令，取消对某个路径上消息事件的监听。如果客户端给出的路径之前就没有被这个客户端订阅，那么这个命令执行无效。
- **MESSAGE 信息：**当客户端在某个订阅的位置收到消息时，这个消息将通过 MESSAGE 关键字进行描述。类似以下信息就是从代理端拿到的消息描述：

```
MESSAGE
redelivered:true
message-id:ID:localhost-34450-1457321490460-4:24:-1:1:1
destination:/test
timestamp:1457331607873
expires:0
priority:4

2345431457331607861
```

- **BEGIN 开始事务命令：**Stomp 协议支持事务模式，在这种模式下，使用 Send 命令从某个客户端发出的消息，在没有使用 COMMIT 命令正式提交前，这些消息不会真正发送给 Stomp 代理端。BEGIN 命令就是用于开启事务。注意，一个事务中可以有一条消息，也可以有多条消息。
- **COMMIT 提交命令：**当完成事务中的信息定义后，使用该命令提交事务。只有使用 COMMIT 命令后，在某一个事务中的一条或者多条消息才会进入 Stomp 代理端的队列，订阅了事件的其他客户端才能收到这些消息。
- **ABORT 取消/终止事务命令：**很明显，这个命令用于取消/终止当前还没有执行 COMMIT 命令的事务。
- **ACK 确认命令：**当客户端使用 SUBSCRIBE 命令进行订阅时，如果在 SUBSCRIBE 命令中制定 ACK 属性为 client，那么这个客户端在收到某条消息（ID 为××××）后，必须向 Stomp 代理端发送 ACK 命令，这样代理端才会认为消息处理成功了；如果 Stomp

客户端在断开连接之前都没有发送 ACK 命令，那么 Stomp 代理端将在这个客户端断开连接后，将这条消息发送给其他客户端。以下是一个 ACK 确认命令的结构示例：

```
ACK
id:MESSAGE ID

^@
```

请注意 head 部分的 id 属性，传递的 id 属性是之前收到的 MESSAGE 信息的 id 值。

- **NACK 不确认命令：**同样是以上的 SUBSCRIBE 命令的状态下，如果这时 Stomp 客户端向 Stomp 代理端发送 NACK 信息，刚证明这条消息在这个客户端处理失败。Stomp 代理端将会把这条消息发送给另一个客户端（无论当前的客户端是否断开连接）。
- **DISCONNECT 断开命令：**这个命令将断开 Stomp 客户端与 Stomp 代理端的连接。

### 9.2.3 MQTT 协议

MQTT (Message Queuing Telemetry Transport, 消息队列遥测协议)。目前协议最新的版本号为 Version 3.1。以下引用文字来源于 MQTT 协议官网上给出的准确定义：

MQ Telemetry Transport (MQTT) is a lightweight broker-based publish/subscribe messaging protocol designed to be open, simple, lightweight and easy to implement. These characteristics make it ideal for use in constrained environments, for example, but not limited to:

Where the network is expensive, has low bandwidth or is unreliable

When run on an embedded device with limited processor or memory resources

MQTT 协议是一个轻量级协议，被设计用于基于代理的发布/订阅消息的场景。MQTT 协议的定义非常简单，并且单次传输的数据量非常少，这使它可以应用于一些特定的受限场景下，例如：

- 网络设施昂贵且网络带宽狭窄、网络稳定性较差的环境，例如卫星通信。
- 嵌入式设备的通信环境。

MQTT 协议由 IBM 主导设计（<http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html>），并有可能成为物联网的重要组成部分。该协议支持所有平台，几乎可以把所有联网物品和外部连接起来，所以常被用来当作物联网中传感器和控制器的通信协议。

MQTT 协议主要由三部分组成：固定的消息头、可变消息头和有效载荷（消息内容体）。其中固定的消息头是每一个 MQTT 消息必须有的，而是否有可变消息头和有效载荷则根据情

况而定。

### 1. 固定消息头

MQTT 协议之所以只需要占用很小的带宽资源，是因为它必须携带的固定消息头最小只有两个 byte，也就是 16 位，最多的时候也不超过 5 个 byte，其中 4 个 byte 都用于记录后续消息体的长度。协议的设计人员充分利用了这 16 个 bit 位描述了尽可能多的消息状态，如图 9-5 所示（来源 MQTT 官网）。

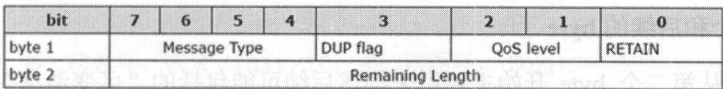


图 9-5 MQTT 固定消息头

本书并不会将 MQTT 协议的所有细节都进行介绍，实际上 MQTT 协议最关键的部分就是它的固定消息头部分，而可变消息头和有效载荷部分的内容都会因为固定消息头中内容的不同而受到影响。

#### (1) 第一个 byte

- Message Type 表示消息类型/消息指令，它占用 4 个 bit，也就是说可以表达最多 16 种不同的消息类型。具体如图 9-6 所示。

Mnemonic	Enumeration	Description
Reserved	0	Reserved
CONNECT	1	Client request to connect to Server
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received (assured delivery part 1)
PUBREL	6	Publish Release (assured delivery part 2)
PUBCOMP	7	Publish Complete (assured delivery part 3)
SUBSCRIBE	8	Client Subscribe request
SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	PING Request
PINGRESP	13	PING Response
DISCONNECT	14	Client is Disconnecting
Reserved	15	Reserved

图 9-6 Message Type 表达形式

- DUP flag：该标记用于客户端和服务端重新交付 PUBLISH、PUBREL、SUBSCRIBE 或



UNSUBSCRIBE 类型的消息，占用一个 bit。如果该标记被使用，那么后续将要说明的 QoS level 范围的值一定大于 0，并且这条消息也需要被 ACK。另外，当这个标记被设置时，则后续介绍的“可变消息头”中将会记录一个 Message ID。

- QoS level: 该标记占用 2 个 bit，用来表示这条消息的传输可靠性级别。包括：0 表示最多传输一次；1 表示至少传输一次；2 表示刚好传输一次；3 为保留数字。这就是为什么只有 QoS level 不为 0 的情况下，DUP flag 标记才会起作用。
- RETAIN: 保留位，MQTT 协议第一个 byte 的最右侧一个 bit 为保留位。

(2) 第二个和后续的 byte

固定消息头从第二个 byte 开始主要用来描述后续可能包括的“可变消息头”+“有效载荷”的总长度（称之为 Remaining Length）。但如果只有一个 byte 来记录总长度显然有些不够，因为一个 byte 为 8 位，它最多能记录消息剩余的 256 长度的内容，但是很多时候后续的消息长度都会超过这个 256 的记录上限，那么怎么办呢？

首先 MQTT 允许最多在固定消息头部分采用 4 个 byte 记录“可变消息头”+“有效载荷”的总长度，另外每一个 byte 的最高位都表示还有没有后续的 byte 记录更大的长度，所以每一个 byte 的最高位都不参与总长度计算。在官方文档中给出了一个表格，来说明当 byte 个数不一样时 Remaining Length 所能表达的最小和最大值。这里我们使用算法示例 + 图解的方式对 MQTT 协议中的 Remaining Length 进行说明：因为本书认为这个算法很有启发性。我们先来看看 MQTT 官网对 Remaining Length 和 byte 个数关系的描述表格，如图 9-7 所示。

Digits	From	To
1	0 (0x00)	127 (0x7F)
2	128 (0x80, 0x01)	16 383 (0xFF, 0x7F)
3	16 384 (0x80, 0x80, 0x01)	2 097 151 (0xFF, 0xFF, 0x7F)
4	2 097 152 (0x80, 0x80, 0x80, 0x01)	268 435 455 (0xFF, 0xFF, 0xFF, 0x7F)

图 9-7 Remaining Length 和 byte 个数的关系

以上表格表示，MQTT 协议的固定头部分可以记录“可变消息头”+“有效载荷”的总长度和记录位数之间的关系，其中 Digits 列表示固定头部分用于记录总长度的 byte 字节数量，最多可以使用 4 个 byte（也即是 4 个字节）。

这个表格初看很好理解，例如表格上说当有 1 个 byte（也就是固定中 Remaining Length 部分 byte 个数为 1）参与记录总长度时，可记录的 MQTT 消息的总长度在 0 到 127 之间（甚至表格中还列举了其 16 进制表示）；当有 2 个 byte 参与记录总长度时，可记录的 MQTT 消息的长度在 128 到 16383 之间。但是细细一看可能有的读者就要觉得不对了：当 byte 数为 2 时一共有 16 个 bit，那么最大的二进制值应该是“11111111 11111111”，换算成 10 进制数值应该是

65535，那么会是 16383 吗？

这个原因在上文其实已经回答了，每一个 byte 有 1 位要用来表示是否还有下一个 byte 共同记录总长度，那么每多一个 byte 都会有 1 位不参与运算。所以实际上 byte 值为 2 时，能够使用的 bit 就只有 14 位而不是 16 位，二进制的“1111111 1111111”就等于 10 进制的 16383。

- 以下是将 10 进制的值转换为 byte（7bit）存储值的代码片段：

```
// 我们带入一个值 X=394，走读执行效果
.....
do
    // 第一次循环，取余 digit=10；第二次循环，digit = 3
    digit = X MOD 128
    // 第一次循环，X 在除以 128 后为 X = 3；第二次循环 X = 0
    X = X DIV 128
    // 如果条件成立，则说明还需要更多的 byte 空间，于是将当前 byte 的最高位标为 1
    // （通过“或”运算完成）
    // if there are more digits to encode, set the top bit of this digit
    if ( X > 0 )
        // 这里只会第一次循环中进入 1 次
        // 当 digit=10 时，它和二进制的 1000 0000 或运算后
        // 得到 digit = 138，二进制表示为 10001010
        digit = digit OR 0x80
    endif
    'output' digit
while ( X> 0 )
.....
```

以上代码片段中已经给出了很清楚的注释，这里就不再对执行过程进行复述了。当带入 X = 394 时，循环将进行两次，两次的输出分别是“138”和“3”。也就是说 MQTT 协议的 Remaining Length 部分会用两个 byte 存储信息，如图 9-8 所示。

bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	1000 1010							
byte 3	0000 0011							

图 9-8 X=394 时存储效果

byte 2 中最高位为“1”，表示其下还有一个 byte 继续描述 Remaining Length。那么 MQTT 信息的接收者在拿到信息后，怎么来解析这些用来表示 Remaining Length 的若干个 byte 呢？请看对下文转换代码的注释描述。

- 以下是将 byte—7bit 存储的值转换为 10 进制值的代码片段：

```

.....
// multiplier 表示当前计算的倍数
// 根据代码片段, 第 1 个 byte 中的倍数为 1, 第二个 byte 中的倍数为 128,
// 第三个 byte 的倍数为 128×128, 第四个 byte 的倍数为 128×128×128
multiplier = 1
value = 0
do
    // 注意这个 digit 不是一个字符串
    // 而是表示依次取 MQTT 中记录 Remaining Length 的每一个 byte
    // 第一次带入的值为 138 (10001010)
    // 第二次带入的值为 10 (00001010)
    digit = 'next digit from stream'
    // 累加最后的输出值, 第一次循环计算结果为 10
    // 第二次循环计算结果为 394
    value += (digit AND 127) * multiplier
    // 每循环一次倍数×128
    multiplier *= 128
// 循环条件是, 当前 byte 的最高位为 1
// 换句话说, 就是当前 byte 的最高位标识了还有后续的 byte 记录 Remaining Length
while ((digit AND 128) != 0)
.....

```

以上代码片段的两次带入结果分别是“10001010”和“00001010”。

## 2. 可变消息头和有效载荷

可变消息头相对于固定消息头来说, 就要简单多了。至少其中的属性不需要像 Remaining Length 那样进行计算。可变消息头的内容很明显是不固定的, 当固定消息头中的 Message Type 属性有不同表达时, 会影响可变头的消息结构。

有效消息载荷就更简单了, 基本就是你所要传输业务信息的二进制表示。其存在形式也受到固定头中 Message Type 属性的影响, 也受到可变消息头内容的影响。例如只有当 Message Type 属性为 CONNECT/SUBSCRIBE/SUBACK/PUBLISH 时, 才允许有消息体的存在。可变消息头和有效载荷的详细信息就不再铺开来讲了, 如果各位读者在业务系统中需要使用 MQTT 协议, 则可以参考官方网站上更详细的介绍。

## 9.2.4 AMQP 协议

AMQP 协议的全称是 Advanced Message Queuing Protocol (高级消息队列协议)。目前 AMQP 协议的版本为 Version 1.0, 这个协议标准在 2014 年通过了国际标准组织 (ISO) 和国际电工委员会 (IEC) 的投票, 成为新的 ISO 和 IEC 国际化标准。目前支持 AMQP 的软件厂商如图 9-9 所示。



图 9-9 AMQP 支持厂商

首先要说明的是目前国内多个技术站点，详细介绍 AMQP 消息格式的文章本来就不多（不包括那些聊聊几笔的转发），而且基本上都没有详细讲解格式本身，只是粗略地说明了 AMQP 消息采用二进制格式。有的文章还向读者传递了有一定误导性的信息，例如说 AMQP 消息格式包括两部分：消息头和消息正文。这是不完整的，虽然 AMQP 消息格式确实包括 Header（消息头）和 Body（消息正文）部分，但是绝对不止这两个部分。

实际上 AMQP 协议要比 Stomp 协议复杂得多，作为一种网络通信协议，AMQP 工作在七层/五层网络模型的应用层，是一个典型的应用层协议；另外，由于 AMQP 协议存在多种元素定义，且这些元素定义工作在不同的领域。例如 Channel 的定义是为了基于网络连接记录会话状态；Queue 等元素帮助 AMQP 完成路由规则，这些元素在 Message 消息记录中都需要有所体现。

所以 AMQP 协议首先要记录网络状态和会话状态，格式如图 9-10 所示（AMQP 帧的定义在 OASIS Advanced Message Queueing Protocol (AMQP) Version 1.0 文档的第 38 页）。

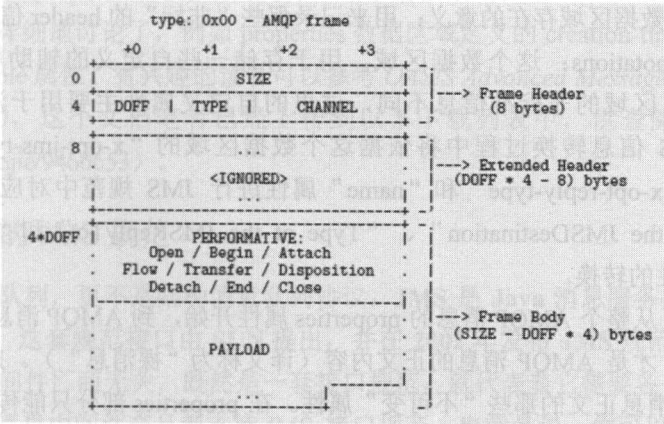


图 9-10 AMQP 协议结构



其中非 PAYLOAD 部分，在网络协议的应用层说明 Channel 的工作状态（当然还有说明整个 AMQP 消息的长度区域：SIZE），我们真正需要的内容存在 PAYLOAD 区域。PAYLOAD 区域（官方译文称为“交付区”，但实际上也可以称为有效载荷）的格式如图 9-11 所示（可以在 *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0* 文档的第 3 部分：Messaging，第 82 页找到详细说明）。

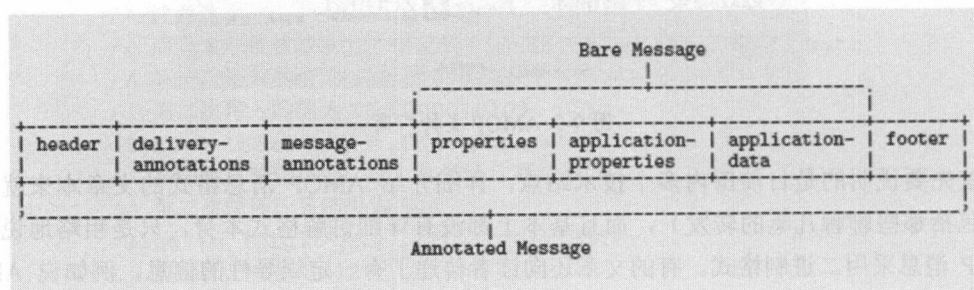


图 9-11 Messaging

在 PAYLAOD 区域一共包含 7 个数据区域：header、delivery-annotations、message-annotations、properties、application-properties、application-data、footer。这些元素的作用如下。

- **header:** header 部分记录了 AMQP 消息的在“支持 AMQP 协议的消息中间件”中的交互状态。例如该条消息在节点间被交互的总次数、优先级、TTL（Time To Live）值等信息。
- **delivery-annotations:** 在 header 部分只能传递规范的、标准的、经过 ISO/IEC 组织定义的属性。那么如果需要在 header 部分传递一些非标准信息怎么办呢？这就是 delivery-annotations 数据区域存在的意义：用来记录那些“非标”的 header 信息。
- **message-annotations:** 这个数据区域，用于存储一些自定义的辅助属性。和 delivery-annotations 区域的非标准信息不同，这里的自定义属性主要用于消息的转换。例如 AMQP-JMS 信息转换过程中将依据这个数据区域的“x-opt-jms-type”、“x-opt-to-type”、“x-opt-reply-type”和“name”属性进行 JMS 规范中对应的“JMSType”、“Type of the JMSDestination”、“Type of the JMSReplyTo”和“JMS\_AMQP\_MA\_name”属性的转换。
- **properties:** 从整个 AMQP 消息的 properties 属性开始，到 AMQP 消息的 application-data 部分结束，才是 AMQP 消息的正文内容（译文称为“裸消息”）。Properties 属性记录了 AMQP 消息正文的那些“不可变”属性。在 properties 部分只能传递规范的、标准的、经过 ISO/IEC 组织定义的属性。例如：消息 id、分组 id、发送者 id、内容编码等。以下是 AMQP 协议文档中对 properties 部分属性的描述（只能包含这些信息）：



```

<type name="properties" class="composite" source="list" provides=
"section">
  <descriptor name="amqp:properties:list" code="0x00000000:
0x00000073"/>
  <field name="message-id" type="*" requires="message-id"/>
  <field name="user-id" type="binary"/>
  <field name="to" type="*" requires="address"/>
  <field name="subject" type="string"/>
  <field name="reply-to" type="*" requires="address"/>
  <field name="correlation-id" type="*" requires="message-id"/>
  <field name="content-type" type="symbol"/>
  <field name="content-encoding" type="symbol"/>
  <field name="absolute-expiry-time" type="timestamp"/>
  <field name="creation-time" type="timestamp"/>
  <field name="group-id" type="string"/>
  <field name="group-sequence" type="sequence-no"/>
  <field name="reply-to-group-id" type="string"/>
</type>

```

- **application-properties**: “应用数据”属性，在这部分数据中主要记录和应用有关的数据，AMQP 的实现产品（例如 RabbitMQ）需要用这部分数据决定其处理逻辑。例如：将消息送入到哪一个 Exchange、消息的 Routing 值是什么、是否进行持久化等。
- **application-data**: 使用二进制格式描述的 AMQP 消息的用户部分内容。即是我们发送出去的真实内容。
- **footer**: 一般在这个数据区域存储辅助内容，例如消息的哈希值、HMAC、签名或者加密细节。

以上是一个 AMQP 消息的完整结构简介。由于篇幅限制在某一个数据区域的“标准”属性就没有再进行更详细地讨论了，例如 properties 数据区域定义的 creation-time 属性、header 数据区域定义的 durable 属性。有兴趣的读者可以参考 *OASIS Advanced Message Queueing Protocol (AMQP) Version 1.0*，这个文档笔者已经上传到以下下载列表中，供查阅（<http://download.csdn.net/detail/yinwenjie/9460653>）。

## 9.2.5 不得不提的 JMS 规范

JMS 不是消息队列，更不是某种消息队列协议。**JMS 是 Java 消息服务接口**，是一套规范的 Java API 接口。这套规范接口由 SUN 提出，并在 2002 年发布 JMS 规范的 Version 1.1 版本。JMS 和消息中间件厂商无关，既然是一套接口规范，就代表着它需要各个厂商进行实现。好消息是，大部分消息中间件产品都支持 JMS 接口规范。也就是说，你可以使用 JMS API 来连接支持 Stomp 协议的 MQ 产品（例如 ActiveMQ）。就像你可以使用 JDBC API 来连接

Oracle 或者 MySQL 一样。

部分网络资料介绍 JMS 是一个消息队列组件，这个说法是错误的。难道你能说 JDBC 是数据库？

当然，这些具体实现 JMS 规范的 Java API 都是由具体的中间件厂商提供的。下面一段代码演示了如何使用 JMS 建立与 ActiveMQ 的连接（本章节后续的内容还会涉及 JMS 的相关内容）：

```
.....
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory;
// 测试使用 JMS API 连接 ActiveMQ
public class JMSProducer {
    // 由于是测试代码，这里忽略了异常处理
    // 正式代码可不能这样做
    public static void main (String[] args) throws Exception {
        // 定义 JMS-ActiveMQ 连接信息
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616", "username", "password");
        Session session = null;
        Destination sendQueue;
        Connection connection = null;
        //进行连接
        connection = connectionFactory.createConnection();
        connection.start();
        //建立会话
        session = connection.createSession(true, Session.SESSION_TRANSACTED);
        //建立 Queue（当然如果有了就不会重复建立）
        sendQueue = session.createQueue("");
        //建立消息发送者对象
        MessageProducer sender = session.createProducer(sendQueue);
        TextMessage outMessage = session.createTextMessage();
        outMessage.setText("这是发送的消息内容");
        //发送（JMS 是支持事务的）
        sender.send(outMessage);
        session.commit();
        //关闭
        sender.close();
    }
}
```

```
connection.close();
connectionFactory.close();
}
```

从以上代码片段的“import”区域可以看到，JMS是由Java语言提供的原生支持。本书再给出一份官方文档链接 <http://activemq.apache.org/amqp.html>，这个官方文档用于描述 ActiveMQ 消息中间件中实现的 AMQP 协议信息转换为 JMS 服务接口能够识别的数据信息（请仔细理解这句话粗体字部分的文字描述）。

## 9.3 MQ 实践：ActiveMQ 基本概念和使用

本节之前的内容从理论层面上为大家介绍了消息协议的基本定义，并花了一定篇幅向读者简要介绍了四种典型的消息协议：XMPP 协议、Stomp 协议、MQTT 协议和 AMQP 协议。本节开始，我们将基于之前的知识点讲解这些协议在具体的“消息队列服务”中是如何被我们操作的。由于笔者在实际工作中经常使用 ActiveMQ，所以就选取这个消息队列服务组件/软件进行讲解。

ActiveMQ 是 Apache 软件基金会的开源产品，支持 AMQP 协议、MQTT 协议（和 XMPP 协议作用类似）、Openwire 协议和 Stomp 协议等多种消息协议。并且 ActiveMQ 完整支持 JMS API 接口规范（当然 Apache 也提供多种其他语言的客户端，例如：C、C++、C#、Ruby、Perl）。

### 9.3.1 ActiveMQ 的简易安装过程

在笔者对本节内容进行初次整理时，ActiveMQ 官方发布的稳定版本是 Version 5.13.2（版本号升级很快，不过并不推荐使用最新的版本，而推荐使用稳定版本）。由于 ActiveMQ 的安装很简单，所以这个过程并不值得我们花很大篇幅进行讨论。过程甚至可以概括为：下载→解压→配置环境变量→运行。

- 下载软件：

你可以在 Apache ActiveMQ 的官方站点下载安装包：<https://activemq.apache.org/download-archives.html>。这里我们示例在 CentOS 6.X 版本下 ActiveMQ 的安装过程，所以下载 Linux 下的压缩包即可（<http://www.apache.org/dyn/closer.cgi?path=/activemq/5.13.2/apache-activemq-5.13.2-bin.tar.gz>）。

- 解压软件:

将下载的安装包放置在 root 用户的 home 目录内, 解压即可 (可以根据自己的需要加压到不同的文件路径下)。如下所示:

```
[root@localhost ~]# tar -zxvf ./apache-activemq-5.13.2-bin.tar.gz
```

以上对压缩包的解压操作使用的是 root 用户, 这是为了演示方便。正式环境中还是建议禁用 root 用户, 为 ActiveMQ 的运行专门创建一个用户和用户组。

- 配置环境变量 (不是必须的):

如果只是在测试环境使用 Apache ActiveMQ, 以便熟悉这个消息中间件的特性和使用方式。那么无须对解压后的软件进行任何配置, 所有可运行的命令都在软件安装目录的 ./bin 目录下。为了使用方便, 最好配置一下环境变量, 如下所示 (注意, 根据自己的软件安装位置, 环境变量的设置是不一样的, 请不要盲目粘贴复制):

```
// 设置该次会话的环境变量
[root@localhost ~]# export PATH=/usr/apache-activemq-5.13.1/bin/linux-
x86-64:$PATH;
// 永久设置环境变量
[root@localhost ~]# echo "export PATH=/usr/apache-activemq-5.13.1/bin/
linux-x86-64:$PATH;" >> /etc/profile
```

在 ActiveMQ Version 5.9+ 的版本之后, Apache ActiveMQ 针对操作系统进行了更深入地优化, 所以你可以看到在 ./bin 目录下, 有一个针对 32 位 Linux 运行命令的 ./linux-x86-32 目录和针对 64 位 Linux 运行命令的 ./linux-x86-64 目录, 请按照自己的实际情况进行环境变量设置和命令运行。

- 运行程序

现在你可以在任何目录运行 ActiveMQ 命令了。注意 ActiveMQ 命令一共有 6 个参数 (console | start | stop | restart | status | dump), 启动 Apache ActiveMQ 使用的命令是 activemq start:

```
[root@localhost ~]# activemq start
```

如果启动成功, 就可以在浏览器上访问服务节点在 8161 端口的管理页面了 (例如 <http://localhost:8161>), 如图 9-12 所示。





图 9-12 ActiveMQ 启动效果

单击“Manage ActiveMQ broker”连接，可以进入管理主界面（初始化的用户名和密码都是 admin）。以上就是 Apache ActiveMQ 消息中间件的最简安装和运行方式。在后续内容中，我们会陆续讨论 ActiveMQ 的集群和高性能优化，那时会专门介绍对应的 ActiveMQ 的配置选项和优化问题。

### 9.3.2 ActiveMQ 的其他命令参数

activemq 命令除了 start 参数用于启动 ActiveMQ 程序，还有另外 5 个参数可以使用：console、stop、restart、status、dump。它们代表的意义是：

- stop: 停止当前 ActiveMQ 节点的运行。
- restart: 重新启动当前 ActiveMQ 节点。
- status: 查看当前 ActiveMQ 节点的运行状态。如果当前 ActiveMQ 节点没有运行，那么将返回“ActiveMQ Broker is not running”的提示信息。注意，status 命令只能告诉开发人员当前节点是停止的还是运行的，除此之外不能从 status 命令获取更多的信息。例如，ActiveMQ 为什么创建 Queue 失败？当前 ActiveMQ 使用了多少内存？而要获取这些信息，需要使用以下参数启动 ActiveMQ 节点。
- console: 使用控制台模式启动 ActiveMQ 节点；在这种模式下，开发人员可以调试、监控当前 ActiveMQ 节点的实时情况，并获取实时状态。
- dump: 如果你使用 console 模式运行 ActiveMQ，那么就可以使用 dump 参数，在 console 控制台上获取当前 ActiveMQ 节点的线程状态快照。



### 9.3.3 在 ActiveMQ 中传递 Stomp 消息

既然我们已经讨论过如何安装和运行 ActiveMQ，也讨论了 Stomp 协议的组织结构，为什么不立即动手试一试操作 ActiveMQ 承载基于 Stomp 协议的消息呢？下面使用 ActiveMQ 提供的 Java 客户端（实际上就是 ActiveMQ 对 JMS 规范的实现），向 ActiveMQ 中的 Queue（示例代码中将这个 Queue 命名为 test）发送一条 Stomp 协议消息，然后再使用 Java 语言的客户端从 ActiveMQ 上接收这条消息。

#### 1. 使用 API 向 ActiveMQ 发送 Stomp 协议消息（生产者端）

```
.....
import org.apache.activemq.transport.stomp.StompConnection;
// 消息生产者
public class TestProducer {
    public static void main(String[] args) {
        try {
            // 建立 Stomp 协议的连接
            StompConnection con = new StompConnection();
            Socket so = new Socket("192.168.61.138", 61613);
            con.open(so);
            // 注意，协议版本可以是 1.2，也可以是 1.1
            con.setVersion("1.2");
            // 用户名和密码，这个不必多说了
            con.connect("admin", "admin");
            // 以下发送一条信息
            con.send("/test", "234543" + new Date().getTime());
        } catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }
}
```

#### 2. 使用 API 从 ActiveMQ 接收 Stomp 协议消息（消费者端）

```
.....
import org.apache.activemq.transport.stomp.StompConnection;
import org.apache.activemq.transport.stomp.StompFrame;
public class TestConsumer {
    public static void main(String[] args) throws Exception {
        // 建立连接
        StompConnection con = new StompConnection();
        Socket so = new Socket("192.168.61.138", 61613);
        con.open(so);
        con.setVersion("1.2");
    }
}
```

```

con.connect("admin", "admin");
String ack = "client";
con.subscribe("/test", "client");
// 接收消息
for(;;) {
    StompFrame frame = null;
    try {
        // 注意，如果没有接收到消息，
        // 那么这个消费者线程会停在这里，直到本次等待超时
        frame = con.receive();
    } catch(SocketTimeoutException e) {
        continue;
    }
    // 打印本次接收到的消息
    System.out.println("frame.getAction() = " + frame.getAction());
    Map<String, String> headers = frame.getHeaders();
    String message_id = headers.get("message-id");
    System.out.println("frame.getBody() = " + frame.getBody());
    System.out.println("frame.getCommandId() = " +
frame.getCommandId());
    // 在 ack 是 client 标记的情况下，确认消息
    if("client".equals(ack)) {
        con.ack(message_id);
    }
}
}
}

```

以上分别是使用 ActiveMQ 提供的 Stomp 协议的消息生产端和 Stomp 协议的消息消费端的代码（如果读者不清楚 Stomp 协议的细节，可以参考 9.2 节中相应的内容。请注意在代码片段中，并没有出现任何一个带有 JMS 名称的包或者类——这是因为 ActiveMQ 为 Stomp 协议提供的 Java API 在内部进行了 JMS 规范的封装。你可以查看 `activemq-stomp` 中关于协议转换部分的源代码 `org.apache.activemq.transport.stomp.JmsFrameTranslator` 和其父级接口 `org.apache.activemq.transport.stomp.FrameTranslator` 来验证这件事情。

以下是 Stomp 协议的消费者端的运行效果（在生产者端已经向 ActiveMQ 插入了一条消息之后）：

```

frame.getAction() = MESSAGE
frame.getBody() = 2345431458460073204
frame.getCommandId() = 0
//注意，由于消息体中插入了一个时间戳
//所以你实际运行后的效果并不会和演示程序的输出完全一致

```

### 9.3.4 ActiveMQ 中的 Queue 和 Topics

ActiveMQ 完全实现了对 JMS 的支持，在 ActiveMQ 提供的管理页面上已经看到有两个功能页面：Queue 和 Topic。**Queue** 和 **Topic** 是 JMS 为开发人员提供的两种不同工作机制的消息队列。在 ActiveMQ 官方的解释是：

- Topics

In JMS a Topic implements publish and subscribe semantics. When you publish a message it goes to all the subscribers who are interested - so zero to many subscribers will receive a copy of the message. Only subscribers who had an active subscription at the time the broker receives the message will get a copy of the message.

中文可以译作：JMS-Topic 队列基于“订阅—发布”模式，当操作者发布一条消息后，所有对这条消息感兴趣的订阅者都可以收到它——也就是说这条消息会被复制成多份然后进行分发。只有当前“活动的”订阅者能够收到消息，换句话说如果当前 JMS-Topic 队列中没有订阅者，这条消息将被丢弃（图 9-13）。

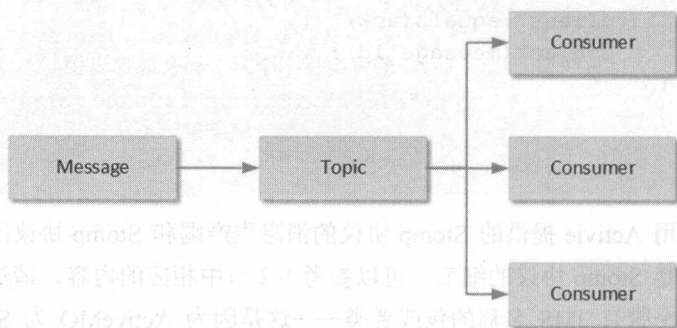


图 9-13 “订阅—发布”模式工作方式

在“订阅—发布”模式下，消息会被复制多份，分别发送给所有“订阅”者。实际上在后文的描述中你将看到，这个复制的过程实际上并没有想象得那样简单。

- Queue

A JMS Queue implements load balancer semantics. A single message will be received by exactly one consumer. If there are no consumers available at the time the message is sent it will be kept until a consumer is available that can process the message. If a consumer receives a message and does not acknowledge it before closing then the message will be

redelivered to another consumer. A queue can have many consumers with messages load balanced across the available consumers.

So Queues implement a reliable load balancer in JMS.

中文可以译作：JMS-Queue 是一种“负载均衡”模式的实现。一个消息能且只能被一个消费者接收。如果当前 JMS-Queue 中没有任何消费者，那么这条消息将会被 Queue 存储起来（实际应用中可以存储在磁盘上，也可以存储在数据库中，完全是看 ActiveMQ 如何配置），直到有一个消费者连接上。另外，当消费者在接收到消息后，如果在它断开与 JMS-Queue 连接之前没有发送 ACK 信息（可以是客户端手动发送，也可以是自动发送），那么这条消息将被发送给其他消费者（图 9-14）。

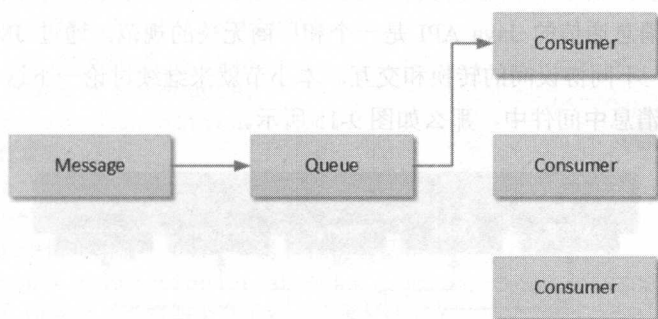


图 9-14 “负载均衡”模式工作方式

在“负载均衡”模式下，一条消息将会发送给我一个消费者，如果当前 Queue 没有消费者，那么消息将进行存储。同样通过后面的介绍，你也会发现这其中的过程并不简单。

表 9-1 摘自互联网上的资料，基本上把 Queue 和 Topic 这两种队列的不同特性说清楚了。

表 9-1

比较项目	Topic 模式队列	Queue 模式队列
工作模式	“订阅—发布”模式：如果当前没有订阅者，那么消息将会被丢弃；如果有多个订阅者，那么这些订阅者都会收到消息	“负载均衡”模式：如果当前没有消费者，则消息也不会丢弃；如果有多个消费者，那么一条消息也只会发送给其中一个消费者，并且要求消费者 ACK 信息
有无状态	无状态	Queue 数据默认会在 MQ 服务器上以文件形式保存，比如 Active MQ 一般保存在 \$AMQ_HOME\data\kr-store\data 下面。也可以配置成 DB 存储

续表

比较项目	Topic 模式队列	Queue 模式队列
传递完整性	如果没有订阅者，则消息会被丢弃	消息不会丢弃
处理效率	由于消息要按照订阅者的数量进行复制，所以处理性能会随着订阅者的增加而明显降低，当然性能下降曲线还要结合不同消息协议自身的性能差异	由于一条消息只发送给一个消费者，所以就算消费者再多，性能也不会有明显降低。当然不同消息协议的具体性能也是有差异的

9.3.5 JMS 和协议间转换

JMS 这套面向消息通信的 Java API 是一个和厂商无关的规范。通过 JMS，我们能实现不同消息中间件厂商、不同协议间的转换和交互。本小节就来继续讨论一下这个问题，如果用一张图来表示 JMS 在消息中间件中，那么如图 9-15 所示。

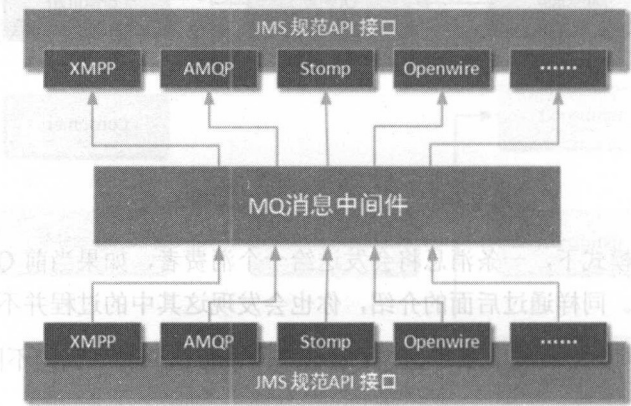


图 9-15 JMS 与消息转换

首先你需要确定正在使用的消息队列中间件实现了 **JMS** 规范，至少应该是大部分实现了。通过 **JMS** 规范，开发人员可以忽略各种消息协议的细节，只要消息在同一队列中，就能够保证各种消息协议间实现互相转换。下面首先来看一个使用 **JMS** API 在 **ActiveMQ** 中操作 **Openwire** 协议消息的简单示例，然后再给出一个通过 **JMS**，实现 **Stomp** 消息协议和 **Openwire** 消息协议间的互转示例。

1. 基于 JMS 的相同协议操作

以下代码使用向某个 Queue（名称为 test）中发送一条消息：



```

.....
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.activemq.ActiveMQConnectionFactory;
// 测试使用 JMS API 连接 ActiveMQ
public class JMSProducer {
    // 由于是测试代码，这里忽略了异常处理
    // 正式代码可不能这样做
    public static void main (String[] args) throws Exception {
        // 定义 JMS-ActiveMQ 连接信息（默认为 Openwire 协议）
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
Factory("tcp://192.168.61.138:61616");
        Session session = null;
        Destination sendQueue;
        Connection connection = null;
        //进行连接
        connection = connectionFactory.createQueueConnection();
        connection.start();
        //建立会话（设置一个带有事务特性的会话）
        session = connection.createSession(true, Session.SESSION_TRANSACTED);
        //建立 Queue（当然如果有了就不会重复建立）
        sendQueue = session.createQueue("/test");
        //建立消息发送者对象
        MessageProducer sender = session.createProducer(sendQueue);
        TextMessage outMessage = session.createTextMessage();
        outMessage.setText("这是发送的消息内容");
        //发送（JMS 是支持事务的）
        sender.send(outMessage);
        session.commit();
        //关闭
        sender.close();
        connection.close();
    }
}

```

当以上代码运行到“start”的位置时，我们可以通过观察 ActiveMQ 管理界面中 connection 列表中的连接信息，发现消息生产者已经建立了一个 Openwire 协议的连接，如图 9-16 所示。

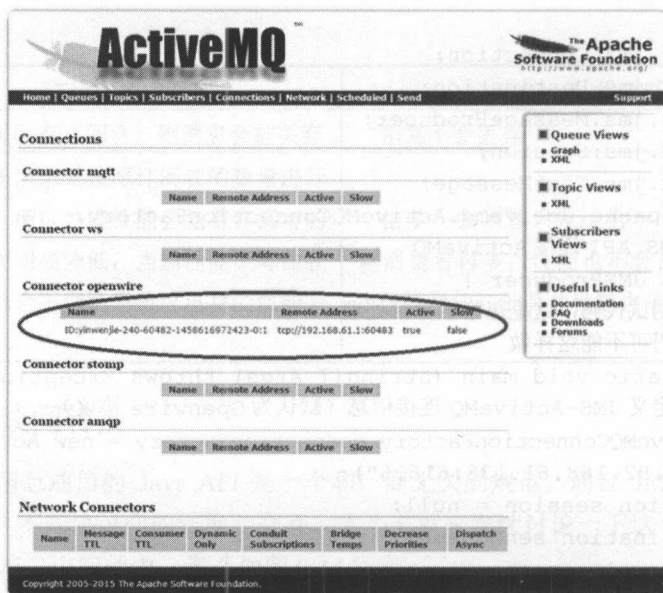


图 9-16 第一个 Openwire 协议连接

从图 9-16 可以确定我们通过 JMS API 建立了一个 Openwire 协议的通信连接。接着使用以下代码，建立一个基于 Openwire 协议连接的“消费者”。注意：消息生产者和消息消费者映射的队列必须一致（在示例代码中，它们都映射名称为 test 的 JMS-Queue）。

- 以下代码使用 JMS 从某个 Queue 中接收消息（其中很多代码片段和发送消息类似）：

```
.....
//测试使用 JMS API 连接 ActiveMQ
public class JMSConsumer {
    // 由于是测试代码，这里忽略了异常处理
    // 正式代码可能不能这样做
    public static void main (String[] args) throws Exception {
        // 定义 JMS-ActiveMQ 连接信息
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://192.168.61.138:61616");
        Session session = null;
        Destination sendQueue;
        Connection connection = null;
        //进行连接
        connection = connectionFactory.createQueueConnection();
        connection.start();
        //建立会话(设置为自动 ACK)
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

```

//建立 Queue（当然如果有了就不会重复建立）
sendQueue = session.createQueue("/test");
//建立消息发送者对象
MessageConsumer consumer = session.createConsumer(sendQueue);
consumer.setMessageListener(new MessageListener() {
    @Override
    public void onMessage(Message arg0) {
        // 接收到消息后，不需要再发送 ACK 了。
        System.out.println("Message = " + arg0);
    }
});
synchronized (JMSConsumer.class) {
    JMSConsumer.class.wait();
}
//关闭
consumer.close();
connection.close();
}
}

```

当以上“消费者”代码运行到 `start` 的位置时，我们通过 ActiveMQ 提供的管理界面可以看到，基于 Openwire 协议的连接增加到了两条，如图 9-17 所示。

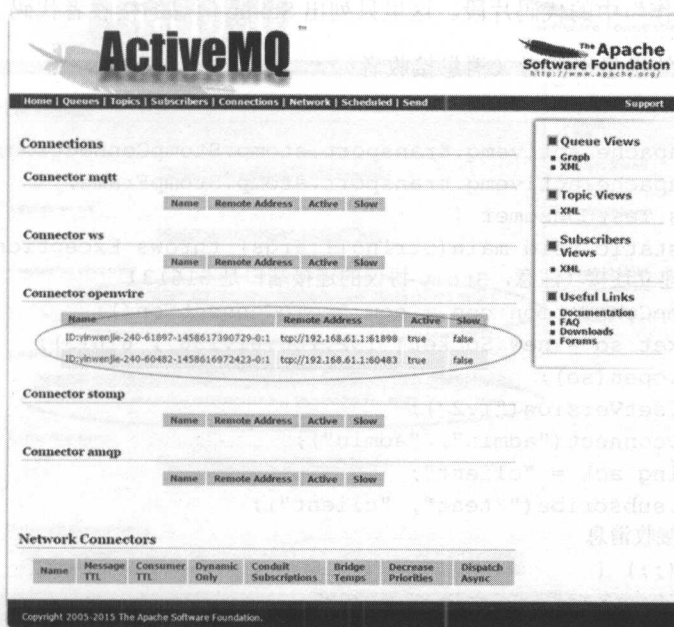


图 9-17 第二个 Openwire 协议连接

注意，你在运行以上测试代码时，不用和示例中的运行顺序完全一致。由于 Queue 模式的队列是要进行消息状态保存的，所以无论你是先运行“消费者”端，还是先运行“生产者”端，最后“消费者”都会收到一条消息。类似如下的效果：

```
Message = ActiveMQTextMessage {commandId = 6, responseRequired = false,
messageId = ID:yinwenjie-240-60482-1458616972423-1:1:1:1:1, originalDestination
= null, originalTransactionId = null, producerId = ID:yinwenjie-240-60482-
1458616972423-1:1:1:1:1, destination = queue:///test, transactionId = TX:ID:
yinwenjie-240-60482-1458616972423-1:1:1:1, expiration = 0, timestamp =
1458617840154, arrival = 0, brokerInTime = 1458617840166, brokerOutTime =
1458617840187, correlationId = null, replyTo = null, persistent = true, type
= null, priority = 4, groupId = null, groupSequence = 0, targetConsumerId =
null, compressed = false, userID = null, content = org.apache.activemq.
util.ByteSequence@66968df8, marshalledProperties = null, dataStructure =
null, redeliveryCounter = 0, size = 0, properties = null, readOnlyProperties
= true, readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer =
false, text = 这是发送的消息内容}
```

## 2. 基于 JMS 的不同协议转换

下面我们将 Openwire 协议的消息通过 JMS 送入 Queue 队列，并且让基于 Stomp 协议的消费者接收到这条消息。为了节约篇幅，基于 Openwire 协议的生产者的代码请参考上面“基于 JMS 的相同协议操作”中的代码片段。这里只列出 Stomp 消息的接收者代码。

### • Stomp 协议的消息消费者（消息接收者）

```
.....
import org.apache.activemq.transport.stomp.StompConnection;
import org.apache.activemq.transport.stomp.StompFrame;
public class TestConsumer {
    public static void main(String[] args) throws Exception {
        // 建立连接（注意，Stomp 协议的连接端口是 61613）
        StompConnection con = new StompConnection();
        Socket so = new Socket("192.168.61.138", 61613);
        con.open(so);
        con.setVersion("1.2");
        con.connect("admin", "admin");
        String ack = "client";
        con.subscribe("/test", "client");
        // 接收消息
        for(;;) {
            StompFrame frame = null;
            try {
                // 注意，如果没有接收到消息，
                // 则这个消费者线程会停在这里，直到本次等待超时
```



```

        frame = con.receive();
    } catch(SocketTimeoutException e) {
        continue;
    }
    // 打印本次接收到的消息
    System.out.println("frame.getAction() = " + frame.getAction());
    Map<String, String> headers = frame.getHeaders();
    String meesage_id = headers.get("message-id");
    System.out.println("frame.getBody() = " + frame.getBody());
    System.out.println("frame.getCommandId() = " + frame.
getCommandId());
    // 在 ACK 是 client 模式的情况下，确认消息
    if("client".equals(ack)) {
        con.ack(meesage_id);
    }
}
}
}

```

当你同时运行 Openwire 消息发送者和 Stomp 消息接收者时，可以在 ActiveMQ 的管理界面看到这两种协议的连接信息，如图 9-18 所示。

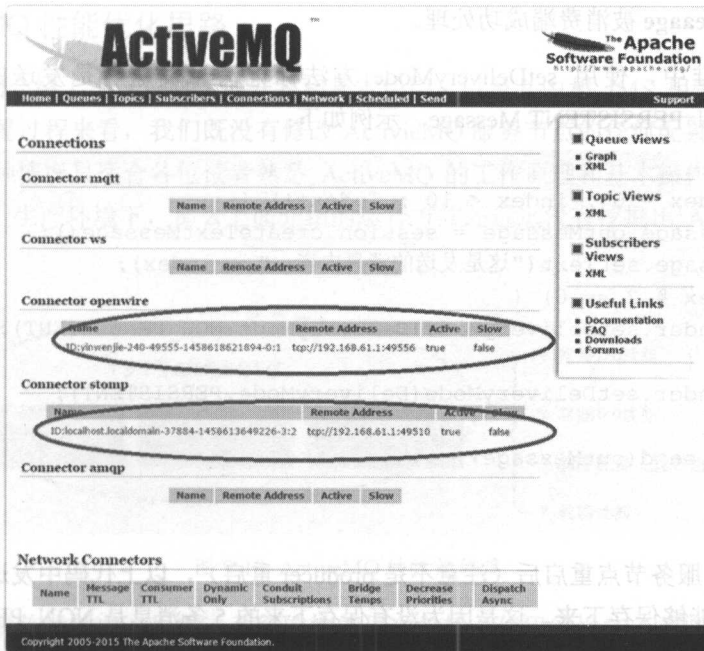


图 9-18 两种不同的协议连接



以下是 Stomp 协议消费者接收到的消息内容（经过转换的 Openwire 协议消息）：

```
frame.getAction() = MESSAGE
frame.getBody() = 这是发送的消息内容
frame.getCommandId() = 0
```

### 9.3.6 持久化消息和非持久化消息

JMS 中对非持久化消息和持久化消息的称呼分别是：NON\_PERSISTENT Message 和 PERSISTENT Message。它们指的是消息在任何一种“发送—接收”模式下（“订阅—发布”模式和“负载均衡”模式），是否进行持久化存储。

NON\_PERSISTENT Message 只存储在 JMS 服务节点的内存区域，不会存储在某种持久化介质上（后文会介绍到 ActiveMQ 可支持的持久化介质有：Kahadb、AMQ、LevelDB 和关系型数据）。在极端情况下，JMS 服务节点的内存区域不够使用了，也只会采用某种辅助方案进行转存（例如 ActiveMQ 会使用磁盘上的一个“临时存储区域”进行暂存。一旦 JMS 服务节点宕机，这些 NON\_PERSISTENT Message 就会丢失。

JMS 中对 PERSISTENT Message 的定义是：这些消息不受 JMS 服务端异常状态的影响，JMS 服务端会使用某种持久化存储方案保存这些消息，直到 JMS 服务端认为这些 PERSISTENT Message 被消费端成功处理。

在 JMS 标准中，使用 setDeliveryMode 方法标记消息发送者是发送的 PERSISTENT Message 还是 NON\_PERSISTENT Message。示例如下：

```
.....
for(int index = 0 ; index < 10 ; index++) {
    TextMessage outMessage = session.createTextMessage();
    outMessage.setText("这是发送的消息内容: " + index);
    if(index % 2 == 0) {
        sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    } else {
        sender.setDeliveryMode(DeliveryMode.PERSISTENT);
    }
    sender.send(outMessage);
}
.....
```

那么当 JMS 服务节点重启后（注意不是 producer 重启），以上代码中发送的 10 条消息只有其中 5 条消息能够保存下来。这是因为没有保存下来的 5 条消息是 NON\_PERSISTENT 形式的消息，所以 ActiveMQ 不会对它们做持久化操作。

### 9.3.7 持续订阅和非持续订阅

持续订阅和非持续订阅，是针对“订阅—发布”模式的细分处理策略，在 JMS 规范中的标准称呼是 Durable-Subscribers 和 Non-Durable Subscribers。

Durable-Subscribers 是指在“订阅—发布”模式下，即使标记为 Durable-Subscribers 的订阅者下线了（可能是因为订阅者宕机，也可能是因为这个订阅者故意下线），“订阅—发布”模式的 Topic 队列也要保存这些消息（视消息不同的持久化策略影响，保存机制不一样），直到下次这个被标记为 Durable-Subscribers 的订阅者重新上线，并正确处理这条消息为止。换句话说，标记为 Durable-Subscribers 的订阅者是否能获得某条消息，和它是否曾经下线没有任何关系。

Non-Durable Subscribers 是指在“订阅—发布”模式下，“订阅—发布”模式的 Topic 队列不用为这些已经下线的订阅者保留消息。当后者将消息按照既定的广播规则发送给当前在线的订阅者后，消息就可以被标记为“处理完成”。

## 9.4 MQ 实践：ActiveMQ 性能优化

### 9.4.1 ActiveMQ 性能优化思路

9.3 节中我们介绍了 ActiveMQ 的安装和基本使用，以及其中和 JMS 相关的概念。从 9.3 节给出的安装配置过程来看，我们既没有修改 ActiveMQ 服务节点的任何配置，也没有采用任何集群方案。这种情况只适合各位读者熟悉 ActiveMQ 的工作原理和基本操作，但是如果要将 ActiveMQ 应用于生产环境下，那么上面介绍的运行方式还远远没有挖掘出 ActiveMQ 的潜在性能（图 9-19）。

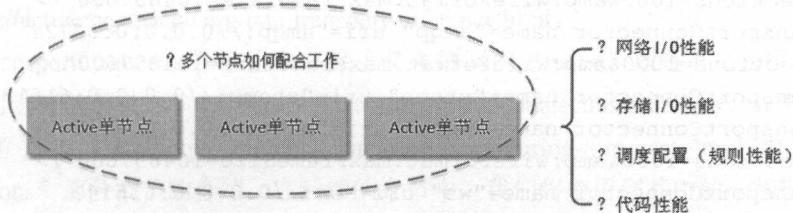


图 9-19 ActiveMQ 的性能维度

根据本书所要陈述的中心思想，系统的性能层次包括：代码级性能、规则性能、存储性能、网络性能，以及多节点协同方法（集群方案），所以我们优化 ActiveMQ 的中心思路也是这样：首先优化 ActiveMQ 单个节点的性能，然后再配置 ActiveMQ 的集群应对集群性能进行整体优

化。下面就按照这个思路，一步步介绍和 ActiveMQ 性能有关的那些事。

在默认情况下，ActiveMQ 的网络信息传递方式基于网络 I/O 模型中的 BIO 方式。那么为了提高 ActiveMQ 单节点的工作性能，首先应该为每一个独立的 MQ 服务节点配置更高效的网络 I/O 模型。然后我们还需要为 ActiveMQ 考虑一种存储方案，让它能高效地完成“持久化”消息的存储操作（也包括对“非持久化”消息的临时存储）。最后，我们还知道使用集群方案能够增加整个软件的性能和稳定性，所以在完成单节点优化以后，还需要提供某种集群方案将多个 ActiveMQ 组合起来，让它们协同工作（一定要协同工作，单纯地安装多个 ActiveMQ 节点而不进行协同，是没法提高性能和稳定性的）。

Apache ActiveMQ 是开源软件，你可以在 ActiveMQ 的官方站点下载到它的源代码，并进行代码级别性能的改造，但很明显这并不是一个能获得高效费比的方式，而且最终导致出现一些莫名其妙的问题。所以笔者不建议更改 ActiveMQ 代码实现。

## 9.4.2 ActiveMQ 中的网络配置

### 1. 基本连接配置

提到了 ActiveMQ 支持多种消息协议，包括 AMQP 协议、MQTT 协议、Openwire 协议、Stomp 协议等。在 ActiveMQ 的官方网站上，列出了目前 ActiveMQ 中支持的所有消息协议，包括：AMQP、MQTT、OpenWire、Stomp、XMPP。

不同的协议需要设置不同的网络监听端口，这个相关设置在 ActiveMQ 安装目录的 ./conf/conf/activemq.xml 主配置文件中。主配置文件采用 XML 格式进行描述，其中的“transportConnectors”标记描述了各种协议的网络监听端口，示例如下：

```
<transportConnectors>
  <transportConnector name="openwire" uri="tcp://0.0.0.0:61616?
maximumConnections=1000&wireFormat.maxFrameSize=104857600"/>
  <transportConnector name="amqp" uri="amqp://0.0.0.0:5672?
maximumConnections=1000&wireFormat.maxFrameSize=104857600"/>
  <transportConnector name="stomp" uri="stomp://0.0.0.0:61613"/>
  <transportConnector name="mqtt" uri="mqtt://0.0.0.0:1883?
maximumConnections=1000&wireFormat.maxFrameSize=104857600"/>
  <transportConnector name="ws" uri="ws://0.0.0.0:61614?
maximumConnections=1000&wireFormat.maxFrameSize=104857600"/>
</transportConnectors>
```

以上配置了 OpenWire 协议的接入端口号为本机所有 IP 设备的 61616（0.0.0.0 代表本机所有 IP 设备）；配置 AMQP 协议的接入端口号为本机所有 IP 设备的 5672；配置 Stomp 协议的接入端口号为本机所有 IP 设备的 61613，等等。这里注意以下几个事实：

- 每一个“transportConnector”标记的 name 属性和 URI 属性都必须填写，name 属性的值可以任意填写，它将作为一个 Connector 元素，显示在 ActiveMQ 管理界面的 Connections 栏目中；
- 每一个“transportConnector”标记的 URI 元素，都有固定写法：URI 头是指定的协议名称，例如 AMQP、MQTT、Stomp 等。然后是 HOST/IP 域名，指定端口监听所在的路由信息；请不要使用 localhost 或者 127.0.0.1 这样的回环地址，否则对应的协议客户端无法通过网络连接到 ActiveMQ；接下来是端口信息，指定的端口不能重复，否则会产生冲突。
- URI 参数部分，每一种协议都有一些特定的参数，读者可参考 ActiveMQ 官网中关于“协议”部分的介绍：<http://activemq.apache.org/protocols.html>。但是有些参数却是各种协议都可共用的，例如以上实例中使用的“maximumConnections”属性，代表这个端口支持的最大连接数量；“wireFormat.maxFrameSize”属性代表支持的“一个完整消息”的最大数据量（单位为 byte）；你可以在 ActiveMQ 官网中对 wire formats 的参数描述中，找到这些默认属性：<http://activemq.apache.org/configuring-wire-formats.html>。

Any transport which involves marshalling messages onto some kind of network transport like TCP or UDP will typically use the OpenWire format. This is configurable to customize how things appear on the wire.

- URI 参数部分，各协议可共用的参数还包括“基本连接特性”相关的参数，这些参数说明可参见官网：<http://activemq.apache.org/connection-configuration-uri.html>。另外如果使用的是 TCP 协议，还可以在 URI 参数部分加入 TCP 相关的属性描述，参见官网 <http://activemq.apache.org/tcp-transport-reference.html>；如果使用的是 UDP 协议（不推荐），那么还可以在 URI 参数部分加入 UDP 相关的属性描述，参见官网 <http://activemq.apache.org/udp-transport-reference.html>。
- “transportConnector”标记中，除了必须填写的“name”属性和“uri”属性，还有一些可选择的属性，例如：enableStatusMonitor、updateClusterClients。详细的属性介绍可参考官方文档 <http://activemq.apache.org/configuring-transports.html> 中“Server side options”部分章节的介绍。在后续的章节中，我们将使用到其中的一些设置项。

## 2. 特别说明

- 从上文给出的配置信息可以发现我们在描述各种消息协议时，URI 描述信息的头部都是采用协议名称：例如描述 AMQP 协议的监听端口时，采用的 URI 描述格式为“amqp:// ……”；描述 Stomp 协议的监听端口时，采用的 URI 描述格式为



“stomp://……”。唯独在进行 OpenWire 协议描述时，URI 头却采用 “tcp://……”。这是因为 **ActiveMQ** 中默认的消息协议就是 **OpenWire**：

OpenWire is binary protocol designed for working with Message Oriented Middleware. It is the native wire format of ActiveMQ.

OpenWire is our cross language Wire Protocol to allow native access to ActiveMQ from a number of different languages and platforms. The Java OpenWire transport is the default transport in ActiveMQ 4.x or later.

- 上文中我们还讲到 ActiveMQ 完整支持 AMQP 协议。但是读者会发现 ActiveMQ 中并没有类似 RabbitMQ 中为 AQMP 协议提供的 Exchange 结构，这是怎么回事呢？实际上在国际标准组织（ISO）和国际电工委员会（IEC）制定的 AMQP Version 1.0 规范文档中（*OASIS Advanced Message Queueing Protocol (AMQP) Version 1.0*），并没有说 AMQP 消息必须经过 Exchange 规则才能够到达队列，也没有规定 Exchange 必须要实现某种规则的路由。所以在支持 AMQP 协议时，是否需要 Exchange 这样的路由处理规则，完全取决于 AMQP 的消息中间件厂商自己的设计。下面一段代码是使用 JMS API 连接 ActiveMQ 的 AMQP 端口，发送 AMQP 消息的示例：

```
.....
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.qpid.amqp_1_0.jms.impl.ConnectionFactoryImpl;

public class JMSProducer {
    public static void main(String[] args) throws Throwable {
        // 注意，JMS-AMQP 使用的是 Apache QPID 的实现。
        // 如果需要运行这段代码，那么请导入 QPID 的客户端
        /*
         * <dependency>
         *   <groupId>org.apache.qpid</groupId>
         *   <artifactId>qpid-amqp-1-0-client-jms</artifactId>
         *   <version>0.32</version>
         * </dependency>
         * */
        ConnectionFactoryImpl factory = ConnectionFactoryImpl.createFromURL(
            "amqp://192.168.61.138:5672");
        Connection connection = factory.createQueueConnection();
        connection.start();
    }
}
```



```

//建立会话，连接到叫作/test 的 Queue 上
Session session = connection.createSession(false, Session.AUTO_
ACKNOWLEDGE);
Destination queue = session.createQueue("/test");
MessageProducer messageProducer = session.createProducer(queue);
//开始发送消息
TextMessage outMessage = session.createTextMessage();
outMessage.setText("23456656457567456");
messageProducer.send(outMessage);
//关闭
messageProducer.close();
connection.close();
}
}
.....

```

### 3. 网络 I/O 模型优化

各位读者是否觉得上文“基本连接配置”中那样的连接端口配置太过冗长，不好进行管理？确实是这样，并且在实际工作中我们也只会使用几种固定的协议。所以 ActiveMQ 在 Version 5.13.0+ 版本后，将 OpenWire、STOMP、AMQP、MQTT 这四种主要协议的端口监听进行了合并，并使用 auto 关键字进行表示。也就是说，ActiveMQ 将监听这一个端口的消息状态，并自动匹配合适的协议格式。配置如下：

```

<transportConnectors>
  <transportConnector name="auto" uri="auto://0.0.0.0:61617? maximum
Connections= 1000" />
</transportConnectors>

```

以上的 URI 配置信息中，可以使用所有通用的 Connection Configuration、Wire Formats Configuring、Server side options 和 TCP Transport Configuration 配置项。但是这种优化只是让 ActiveMQ 的连接管理变得简洁了，并没有提升单个节点的处理性能。如果不特别指定 ActiveMQ 的网络监听端口，那么这些端口都将使用 BIO 网络 I/O 模型。所以为了首先提高单节点的网络吞吐性能，我们需要明确指定 Active 的网络 I/O 模型，如下所示：

```

<transportConnectors>
  <transportConnector name="nio" uri="nio://0.0.0.0:61618? maximumCon
nections=1000"/>
</transportConnectors>

```

请注意，URI 格式头以“nio”开头，表示这个端口使用以 TCP 协议为基准的多路复用网络 I/O 模型（NIO）。但是这样的设置方式，只能使这个端口支持 OpenWire 协议。那么我们怎么既让这个端口支持多路复用网络 I/O 模型，又让它支持多个协议呢？ActiveMQ 的服务端

设置，允许开发人员使用“+”符号来为端口设置多种特性，如下：

```
// 表示这个端口使用 NIO 模型支持 Stomp 协议
<transportConnector name="stomp+nio" uri="stomp+nio://0.0.0.0:61613?transport.
transformer=jms"/>
// 表示这个端口支持 AMQP 协议和 SSL 密文传输
<transportConnector name="amqp+ssl" uri="amqp+ssl://localhost:5671"/>
```

所以如果我们既需要某一个端口支持多路复用网络 I/O 模型，又需要它支持多个协议，那么可以进行如下的配置：

```
<transportConnector name="auto+nio" uri="auto+nio://0.0.0.0:61608? maximum
Connections=1000" />
```

另外，如果是为了生产环境进行的配置，那么还应该配置这个端口支持的最大连接数量、设置每一条消息的最大传输值、设置多路复用网络 I/O 模型使用的线程池最大工作线程数量（你已经知道了这些设置的文档所在位置，所以可以根据自己的情况进行设置属性的增减）：

```
<transportConnector
name="auto+nio"
uri="auto+nio://0.0.0.0:61608?    maximumConnections=1000&wireFormat.
maxFrameSize=104857600&org.apache.activemq.transport.nio.SelectorManager
.corePoolSize=20&org.apache.activemq.transport.nio.SelectorManager.maxim
umPoolSize=50" />
```

图 9-20 是改变网络连接设置后，ActiveMQ 管理控制台中 Connections 页面显示的内容。注意 WS 协议的端口是额外保留的配置——因为 auto 模式中的协议不支持 WS。

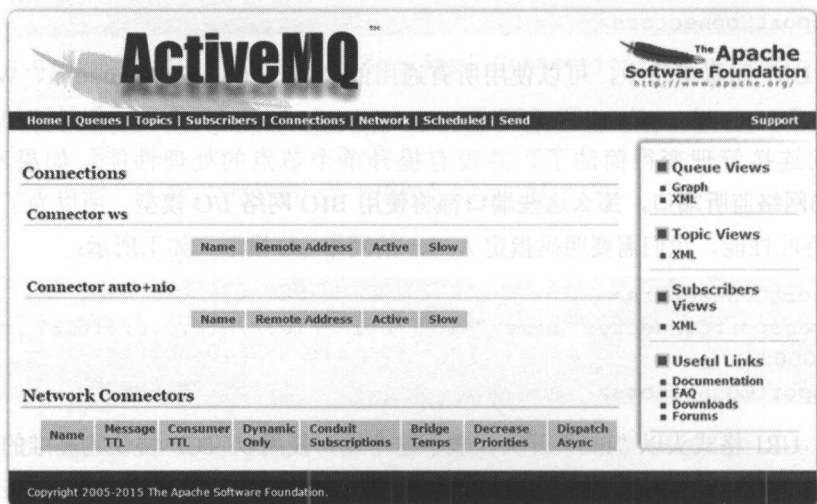


图 9-20 网络设置后的显示内容

### 9.4.3 ActiveMQ 处理规则和优化

在 ActiveMQ 单个服务节点的优化中，除了对 ActiveMQ 单个服务节点的网络 I/O 模型进行优化，生产者发送消息的策略和消费者处理消息的策略也关乎整个消息队列系统是否能够高效工作。请看如图 9-21 所示的消息生产者和消息消费者的简要工作原理图。

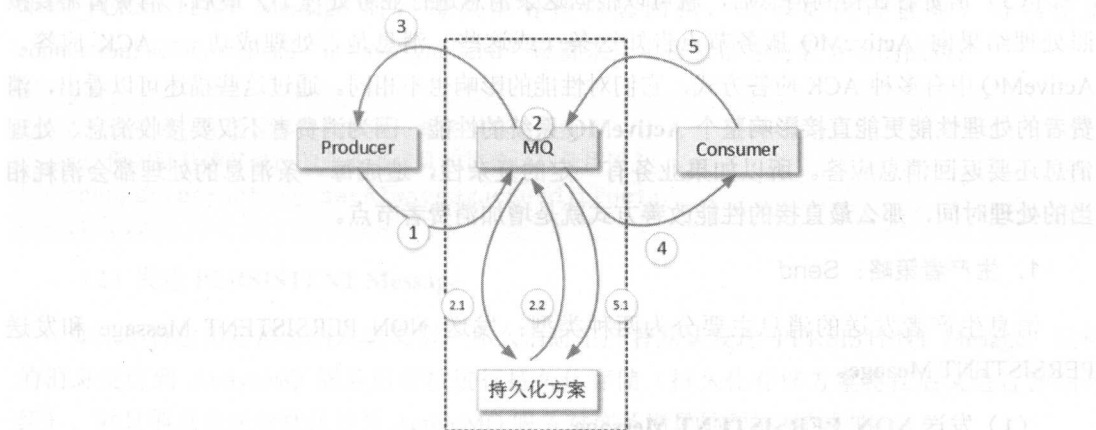


图 9-21 ActiveMQ 消息处理规则

(1) Producer 是消息生产者，作为一个发送消息的客户端它既可以使用同步的消息发送模式，也可以使用异步的消息发送模式。另外，消息生产者在 ActiveMQ 服务节点产生消息堆积的情况下，也不能一味地追求发送效率。还好，这种情况下的消息生产者端有完整的保证机制——Slow Producer。另外，JMS 提供事务功能，所以生产者是否开启事务发送消息，将会影响消息发送性能。

(2) 在整个消息处理规则中，ActiveMQ 服务节点最极端的情况就是产生消息堆积（消息的消费速度持续低于消息生成速度，就会出现消息堆积）。那么 ActiveMQ 服务节点除了网络 I/O 模型的优化，最大的优化点就是：如何应对消息堆积。基本思路是：NON\_PERSISTENT Message 在内存堆积后，转储到 Temp Store 区域（当然也可以设置为不转储）；PERSISTENT Message 无论怎样都会先使用持久化方案存储到永久存储区域，再视情况决定是否进行发送；在这些区域也产生堆积后，通知消息生产者使用 Slow Producer 机制。

(3) ActiveMQ 服务节点在成功完成 PERSISTENT Message 的持久存储操作后，就会向消息生产者发送一个确认信息，表示该条消息已处理完成。如果 ActiveMQ 服务节点接收的是 NON\_PERSISTENT Message，那么生产者默认情况下不会等待服务节点的回执。实际上 PERSISTENT Message 的发送也可以设置为不等待回执，这样可以显著提高生产端的消息发送效率。

(4) ActiveMQ 服务节点会以一种设置好的策略将消息发送给消费者，这个策略的原则是 **ActiveMQ 服务节点主动推送消息给某一个消费者**（不是消费者主动拉取），这样的方式有利于 ActiveMQ 服务节点占据整个策略的领导地位。在这个策略中，**最重要的属性是 prefetchSize**：单次获得的消息数量。除此以外消费者端也可以调整 I/O 网络模型。

(5) 消费者在得到消息后，就可以根据这条消息进行业务处理了。最后，消费者需要按照处理结果向 ActiveMQ 服务节点告知这条（或这些）消息是否处理成功——ACK 应答。ActiveMQ 中有多种 ACK 应答方式，它们对性能的影响也不相同。通过这些描述可以看出，消费者的处理性能更能直接影响整个 ActiveMQ 系统的性能。因为消费者不仅要接收消息、处理消息还要返回消息应答。所以如果业务有一定的复杂性，造成每一条消息的处理都会消耗相当的处理时间，那么最直接的性能改善方式就是增加消费者节点。

### 1. 生产者策略：Send

消息生产者发送的消息主要分为两种类型：发送 **NON\_PERSISTENT Message** 和发送 **PERSISTENT Message**。

#### (1) 发送 NON\_PERSISTENT Message

默认情况下，ActiveMQ 服务认为生产者端发送的都是 **PERSISTENT Message**。所以如果要发送 **NON\_PERSISTENT Message**，那么消息生产者就要明确指定。以下语句明确指定消息发送者将要发送 **NON\_PERSISTENT Message**：

```
.....  
// 设置发送者发送的是 NON_PERSISTENT Message  
MessageProducer sender = session.createProducer(sendQueue);  
sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);  
.....
```

发送 **NON\_PERSISTENT Message** 时，消息发送方默认使用异步方式：即是说消息发送后发送方不会等待 **NON\_PERSISTENT Message** 在服务端的任何回执。那么问题来了：如果这时 ActiveMQ 服务器上已经出现了消息堆积，并且堆积程度已经达到“无法再接收新消息”的极限情况了，那么消息发送方如何知晓并采取相应的策略呢？

实际上所谓的异步发送也并非绝对的异步，消息发送者会在发送若干次 **NON\_PERSISTENT Message** 后等待服务端进行回执，这样消息发送者就能够知道服务器端是否要求自己启用保护机制了（这个配置只是针对使用异步方式进行发送消息的情况）：

```
.....  
// 以下语句设置消息发送者在累计发送 102400byte 大小的消息后（可能是一条消息也可能是多  
// 条消息）  
// 等待服务端进行回执，以便确定之前发送的消息是否被正确处理
```



```
// 另外还可以确定服务器端是否产生了过量的消息堆积，需要减慢消息生产端的生产速度
connectionFactory.setProducerWindowSize(102400);
.....
```

如果使用的是异步发送方式，那么必须通过以上代码指明回执点。例如发送 `NON_PERSISTENT Message` 时，这时消息发送者默认使用异步方式。那么如果想在发送 `NON_PERSISTENT Message` 时，每次都等待消息回执，又该如何设置呢？可以使用 `connectionFactory` 提供的“`alwaysSyncSend`”设置来指定每次都等待服务端的回执：

```
.....
// 设置成：无论怎样每次都等待服务器端的回执
// 但关键是确定自己的业务需求真的需要这样使用吗？
connectionFactory.setAlwaysSyncSend(true);
.....
```

## (2) 发送 PERSISTENT Message

如果不特意指定消息的发送类型，那么消息生产者默认发送 `PERSISTENT Message`。这样的消息发送到 `ActiveMQ` 服务后将被进行持久化存储（持久化存储方案将在后文进行详细介绍），并且消息发送者默认等待 `ActiveMQ` 服务对这条消息处理情况的回执。

以上这个过程非常耗时，`ActiveMQ` 服务端不但要接收消息，在内存中完成存储，并且按照 `ActiveMQ` 服务端设置的持久化存储方案对消息进行存储（主要的处理时间耗费在这里）。为了提高 `ActiveMQ` 在接受 `PERSISTENT Message` 时的性能，`ActiveMQ` 允许开发人员遵从 `JMS API` 中的设置方式，为消息发送端在发送 `PERSISTENT Message` 时提供异步方式：

```
.....
// 使用异步传输
// 上文已经说过，如果发送的是 NON_PERSISTENT Message
// 那么默认就是异步方式
connectionFactory.setUseAsyncSend(true);
.....
```

一旦进行了这样的设置，就需要设置回执窗口：

```
.....
// 同样设置消息发送者在累计发送 102400byte 大小的消息后
// 等待服务端进行回执，以便确定之前发送的消息是否被正确处理
connectionFactory.setProducerWindowSize(102400);
.....
```

## 2. 生产者策略：事务

`JMS` 规范中支持带事务的消息，也就是说可以启动一个事务（并由消息发送者的连接会话设置一个事务号 `Transaction ID`），然后在事务中发送多条消息。这个事务提交前这些消息都



不会进入队列（无论是 Queue 还是 Topic）。

不进入队列，并不代表 JMS 不会在事务提交前将消息发送给 ActiveMQ 服务端。实际上这些消息都会发送给服务端，ActiveMQ 服务发现这是一条带有 Transaction ID 的消息，就会首先把这条消息放置在“transaction store”区域中（并且带有 redo 日志，这样保证在收到 rollback 指令后能进行取消操作），等待这个 Transaction ID 被回滚（rollback）或者提交（commit）。

一旦这个 Transaction ID 被提交（commit），ActiveMQ 才会依据自身设置的 PERSISTENT Message 处理规则或者 NON\_PERSISTENT Message 处理规则，将 Transaction ID 对应的 Message 进行入队操作。以下代码示例了如何在生产者端使用事务发送消息：

```
.....
//进行连接
connection = connectionFactory.createQueueConnection();
connection.start();
//建立会话（设置一个带有事务特性的会话）
session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
//建立 Queue（当然如果有了就不会重复建立）
Queue sendQueue = session.createQueue("/test");
//建立消息发送者对象
MessageProducer sender = session.createProducer(sendQueue);
//发送（JMS 是支持事务的）
for(int index = 0 ; index < 10 ; index++) {
    TextMessage outMessage = session.createTextMessage();
    outMessage.setText("这是发送的消息内容-----" + index);
    // 无论是 NON_PERSISTENT Message 还是 PERSISTENT Message
    // 都要在 commit 后才能真正地入队
    if(index % 2 == 0) {
        sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    } else {
        sender.setDeliveryMode(DeliveryMode.PERSISTENT);
    }
    // 没有 commit 的消息，也要先发送给服务端
    sender.send(outMessage);
}
session.commit();
.....
```

以上代码中，在“connection.createSession”这个方法中一共有两个参数，第一个布尔型参数很好理解，就是表示这个连接会话是否启动事务；第二个整型参数表示了消息消费者的“应答模型”，本书会在下文“消费者策略：ACK”中进行详细介绍。

### 3. 生产者策略：ProducerFlowControl

生产限流控制，是 ActiveMQ 消息生产者端最为重要的性能策略，它主要设定了 ActiveMQ 服务节点在产生消息堆积，并超过限制大小的情况下，如何进行消息生产者端的限流。

具体来说，如果以上情况在 ActiveMQ 出现，那么当生产者端再次接收 ActiveMQ 的消息回执时，ActiveMQ 就会让消息生产者进入等待状态或者在发送者端直接抛出 `JMSEException` 异常。如果对自己 ActiveMQ 服务端的底层性能和消费者端的性能足够自信，那么也可以配置 ActiveMQ 而不进行 `ProducerFlowControl`。

在 ActiveMQ 的主配置文件 `activemq.xml` 中，关于 `ProducerFlowControl` 策略的控制标签是“`destinationPolicy`”和它的子标签。请看如下配置示例：

```
.....
<destinationPolicy>
  <policyMap>
    <policyEntries>
      <policyEntry topic="*" producerFlowControl="false"/>
    </policyEntries>
  </policyMap>
</destinationPolicy>
.....
```

以上示例配置所有的 Topic 模式的队列不进行 `producerFlowControl` 策略控制。当然还可以为队列配置启用 `producerFlowControl` 策略：

```
.....
<policyEntry queue="*" producerFlowControl="true" memoryLimit="200mb">
</policyEntry>
.....
```

以上配置项表示为 ActiveMQ 中的所有 Queue 模式的队列启用 `producerFlowControl` 策略，并且限制每个 Queue 信息的最大内存存储限制（`memoryLimit`）为 200MB；这是指最多使用 200MB 的内存区域，而不是说 Queue 中消息的总大小为 200MB。例如，在 ActiveMQ 5.X+ 版本中 `NON_PERSISTENT` Message 会被转出到 `temp store` 区域，所以有可能你观察到的现象是，无论怎样堆积 `NON_PERSISTENT` Message 消息，Queue 的使用内存始终无法达到 200MB。

```
.....
<policyEntry queue="*" producerFlowControl="true" memoryLimit="200mb">
  <pendingQueuePolicy>
    <vmQueueCursor/>
  </pendingQueuePolicy>
</policyEntry>
.....
```

```
</policyEntry>
```

```
.....
```

以上配置表示只使用内存存储 Queue 中的所有消息，特别是 NON\_PERSISTENT Message 只存储在内存中，不使用 temp store 区域进行转储。在官方文档中，有关于 policyEntry 标签的所有配置选项都有完整说明：<http://activemq.apache.org/per-destination-policies.html>。

#### 4. 消费者策略：Dispatch Async

讨论完消息生产者的关键性能点，我们再将目光转向消息消费者（接收者端）；比起消息生产者来说消息消费者的性能更能影响 ActiveMQ 系统的整体性能，因为要成功完成一条消息的处理，它的工作要远远多于消息生产者。

在默认情况下，ActiveMQ 服务采用异步方式向消费者端推送消息。也就是说 ActiveMQ 服务端在向某个消费者会话推送消息后，不会等待消费者的响应信息，直到消费者处理完消息后，主动向服务端返回处理结果。如果你对自己的消费者性能足够满意，那么也可以将这个进程设置为“同步”：

```
.....
```

```
// 设置为同步
```

```
connectionFactory.setDispatchAsync(false);
```

```
.....
```

#### 5. 消费者策略：Prefetch

消费者关键策略中，需要重点讨论的是消费者“预取数量”——PrefetchSize。可以想象，如果消费者端的工作策略是按照某个周期（例如 1 秒），主动到服务器端一条一条请求新的消息，那么消费者的工作效率一定是极低的；所以在 ActiveMQ 系统中，默认的策略是 ActiveMQ 服务端一旦有消息，就主动按照设置的规则推送给当前活动的消费者。其中每次推送都有一定的数量限制，这个限制值就是 PrefetchSize。

针对 Queue 工作模型的队列和 Topic 工作模型的队列，ActiveMQ 有不同的默认“预取数量”；针对 NON\_PERSISTENT Message 和 PERSISTENT Message，ActiveMQ 也有不同的默认“预取数量”：

- PERSISTENT Message—Queue: prefetchSize=1000
- NON\_PERSISTENT Message—Queue: prefetchSize=1000
- PERSISTENT Message—Topic: prefetchSize=100
- NON\_PERSISTENT Message—Topic: prefetchSize=32766

ActiveMQ 中设置的各种默认预取数量一般情况下不需要进行改变。如果你使用默认的异步方式从服务器端推送消息到消费者端，且你对消费者端的性能有足够的信心，可以加大预取

数量的限制。但是非必要情况下，请不要设置 `PrefetchSize=1`，因为这样就是一条一条地取数据；也不要设置为 `PrefetchSize=0`，因为这将导致关闭服务器端的推送机制，改为客户端主动请求。

- 可以通过 `ActiveMQ PrefetchPolicy` 策略对象更改预取数量

```
.....
//预取策略对象
ActiveMQ PrefetchPolicy prefetchPolicy = connectionFactory.getPrefetchPolicy();
//设置 Queue 的预取数量为 50
prefetchPolicy.setQueuePrefetch(50);
connectionFactory.setPrefetchPolicy(prefetchPolicy);
//进行连接
connection = connectionFactory.createQueueConnection();
connection.start();
.....
```

- 也可以通过更改 `Properties` 属性（当然还可以加入其他属性）预取数量：

```
.....
Properties props = new Properties();
props.setProperty("prefetchPolicy.queuePrefetch", "1000");
props.setProperty("prefetchPolicy.topicPrefetch", "1000");
//设置属性
connectionFactory.setProperties(props);
//进行连接
connection = connectionFactory.createQueueConnection();
connection.start();
.....
```

## 6. 消费者策略：事务和死信

### (1) 消费者端事务

JMS 规范除了为消息生产者提供事务支持，还为消息消费者准备了事务的支持。你可以通过在消费者操作事务的 `commit` 和 `rollback` 方法，向服务器告知一组消息是否处理完成。采用事务的意义在于，一组消息要么被全部处理并确认成功，要么被全部回滚（`rollback`）并且重新处理。

```
.....
//建立会话（采用 commit 方式确认一批消息处理完毕）
session = connection.createSession(true, Session.SESSION_TRANSACTED);
//建立 Queue（当然如果有了就不会重复建立）
sendQueue = session.createQueue("/test");
//建立消息发送者对象
MessageConsumer consumer = session.createConsumer(sendQueue);
consumer.setMessageListener(new MyMessageListener(session));
```



```

.....
class MyMessageListener implements MessageListener {
    private int number = 0;
    private Session session;
    public MyMessageListener(Session session) {
        this.session = session;
    }
    public void onMessage(Message message) {
        // 打印这条消息
        System.out.println("Message = " + message);
        // 如果条件成立, 就向服务器确认这批消息处理成功
        // 服务器将从队列中删除这些消息
        if(number++ % 3 == 0) {
            try {
                this.session.commit();
            } catch (JMSEException e) {
                e.printStackTrace(System.out);
            }
        }
    }
}

```

以上代码演示的是消费者通过事务 `commit` 的方式, 向服务器确认一批消息正常处理完成的方式。请注意代码示例 “`session = connection.createSession(true, Session.SESSION_TRANSACTED);`” 语句。第一个参数表示连接会话启用事务支持; 第二个参数表示使用 `commit` 或者 `rollback` 的方式进行向服务器应答。

这是调用 `commit` 的情况, 那么如果调用 `rollback` 方法又会发生什么情况呢? 调用 `rollback` 方法时, 在 `rollback` 之前已处理过的消息 (注意, 并不是所有预取的消息) 将重新发送一次到消费者端 (发送给同一个连接会话)。并且消息中 `redeliveryCounter` (重发次数计数器) 属性将会加 1。请看如下所示的代码片段和运行结果:

```

public void onMessage(Message message) {
    // 打印这条消息
    System.out.println("Message = " + message);
    // rollback 这条消息
    this.session.rollback();
}

```

以上代码片段中, 我们不停地回滚正在处理的这条消息, 通过打印出来的信息可以看到, 这条消息被不停地重发:

```

Message = ActiveMQTextMessage {... redeliveryCounter = 0, text = 这是
发送的消息内容-----20}

```



```

Message = ActiveMQTextMessage {... redeliveryCounter = 1, text = 这是
发送的消息内容-----20}
Message = ActiveMQTextMessage {... redeliveryCounter = 2, text = 这是
发送的消息内容-----20}
Message = ActiveMQTextMessage {... redeliveryCounter = 3, text = 这是
发送的消息内容-----20}
Message = ActiveMQTextMessage {... redeliveryCounter = 4, text = 这是
发送的消息内容-----20}

```

可以看到同一条记录被重复处理，并且其中的 `redeliveryCounter` 属性不断累加。

## (2) 重发和死信队列

消息处理失败后，不断重发消息肯定不是最好的处理办法：如果一条消息被不断地处理失败，那么最可能的情况就是这条消息承载的业务内容本身就有问题。无论重发多少次，这条消息还是会处理失败。

为了解决这个问题，ActiveMQ 中引入了“死信队列”（Dead Letter Queue）的概念。即一条消息在被重发了多次后（默认为重发 6 次 `redeliveryCounter=6`），将会被 ActiveMQ 移入“死信队列”。开发人员可以在这个 Queue 中查看处理出错的消息，进行人工干预（图 9-22）。

Queues

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
/test	4551	1	4571	25	Browse Active Consumers Active Producers	Send To Purge Delete
ActiveMQ.DLQ	10	0	5	0	Browse Active Consumers Active Producers	Send To Purge Delete

图 9-22 死信显示效果

默认情况下“死信队列”只接受 PERSISTENT Message 形式的消息，如果 NON\_PERSISTENT Message 形式的消息超过了重发上限，将直接被删除。以下配置信息可以让 NON\_PERSISTENT Message 形式的消息在超过重发上限后，也移入“死信队列”：

```

<policyEntry queue="">
  <deadLetterStrategy>
    <sharedDeadLetterStrategy processNonPersistent="true" />
  </deadLetterStrategy>
</policyEntry>

```

另外，默认重发次数 `redeliveryCounter` 的上限也是可以进行设置的，为了保证消息异常情况下尽可能小地影响消费者端的处理效率，实际工作中建议将这个上限值设置为 3。原因已经说过，如果消息本身的业务内容就存在问题，那么重发多少次也没有用。

```
RedeliveryPolicy redeliveryPolicy = connectionFactory.getRedeliveryPolicy();
// 设置最大重发次数
redeliveryPolicy.setMaximumRedeliveries(3);
```

实际上 ActiveMQ 的重发机制还包括以上提到的 `rollback` 方式在内的多种方式:

- 在支持事务的消费者连接会话中调用 `rollback` 方法。
- 在支持事务的消费者连接会话中, 使用 `commit` 方法明确告知服务器端消息已处理成功前, 会话连接就终止了(最可能是异常终止)。
- 在需要使用 ACK 模式的会话中, 使用消息的 `acknowledge` 方式明确告知服务器端消息已处理成功前, 会话连接就终止了(最可能是异常终止)。

以上几种重发机制有一些小小的差异, 主要体现在 `redeliveryCounter` 属性的作用区域。简而言之, 第一种方法 `redeliveryCounter` 属性的作用区域是本次连接会话, 而后两种 `redeliveryCounter` 属性的作用区域是在整个 ActiveMQ 系统范围。

## 7. 消费者策略: ACK

消费者端, 除了可以使用事务方式来告知 ActiveMQ 服务端一批消息已经成功处理, 还可以通过 JMS 规范中定义的 `acknowledge` 模式来实现同样功能。事实上 `acknowledge` 模式更为常用。

### (1) 基本使用

如果选择使用 `acknowledge` 模式, 那么你至少有 4 种方式使用它, 且这 4 种方式的性能区别很大:

- `AUTO_ACKNOWLEDGE` 方式: 这种方式下, 当消费者端通过 `receive` 方法或者 `MessageListener` 监听方式从服务端得到消息后(无论是 `pul` 方式还是 `push` 方式), 消费者连接会话会自动认为消费者端对消息的处理是成功的。但请注意, 这种方式下消费者端不一定是向服务端一条一条 ACK 消息。
- `CLIENT_ACKNOWLEDGE` 方式: 这种方式下, 当消费者端通过 `receive` 方法或者 `MessageListener` 监听方式从服务端得到消息后(无论是 `pul` 方式还是 `push` 方式), 必须显示调用消息中的 `acknowledge` 方法。如果不这样做, 那么 ActiveMQ 服务器端将不会认为这条消息处理成功:

```
.....
public void onMessage(Message message) {
    //=====
    //这里进行业务处理
    //=====
    try {
```

```

// 显示调用 ACK 方法
message.acknowledge();
} catch (JMSEException e) {
    e.printStackTrace();
}
}
.....

```

- **DUPS\_OK\_ACKNOWLEDGE 方式：**批量确认方式。消费者会按照一定的策略向服务器端间隔发送一个 ACK 标示，表示某一批消息已经处理完成。DUPS\_OK\_ACKNOWLEDGE 方式和 AUTO\_ACKNOWLEDGE 方式在某些情况下是一致的，这个在后文会讲到。
- **INDIVIDUAL\_ACKNOWLEDGE 方式：**单条确认方式。这种方式是 ActiveMQ 单独提供的一种方式，其常量定义的位置都不在 javax.jms.Session 规范接口中，而是在 org.apache.activemq.ActiveMQSession 这个类中。这种方式消费者端将会逐一向 ActiveMQ 服务端发送 ACK 信息。所以这种 ACK 方式的性能很差，除非有特别的业务要求，否则不建议使用。

## (2) 工作方式和性能

笔者建议首先考虑使用 AUTO\_ACKNOWLEDGE 方式确认消息。如果这样做，那么请一定使用 optimizeACK 优化选项，并且重新设置 prefetchSize 数量为一个较小值（因为 1000 条的默认值在这样的情况下就显得比较大了）：

```

.....
//ACK 优化选项（实际上默认情况下是开启的）
connectionFactory.setOptimizeAcknowledge(true);
//ACK 信息最大发送周期(毫秒)
connectionFactory.setOptimizeAcknowledgeTimeOut(5000);
connection = connectionFactory.createQueueConnection();
connection.start();
.....

```

AUTO\_ACKNOWLEDGE 方式的根本意义是“延迟确认”，消费者端在处理消息后暂时不会发送 ACK 标示，而是把它缓存在连接会话的一个 pending 区域，等到这些消息的条数达到一定的值（或者等待时间超过设置的值），再通过一个 ACK 指令告知服务端这一批消息已经处理完成；而 optimizeACK 选项（指明 AUTO\_ACKNOWLEDGE 采用“延迟确认”方式）只有当消费者端使用 AUTO\_ACKNOWLEDGE 方式时才会起效。

“延迟确认”的数量阈值：prefetch×0.65

“延迟确认”的时间阈值：> optimizeAcknowledgeTimeOut

DUPS\_OK\_ACKNOWLEDGE 方式也是一种“延迟确认”策略，如果目标队列是 Queue 模式，那么它的工作策略与 AUTO\_ACKNOWLEDGE 方式是一样的。也就是说，如果这时 **PrefetchSize = 1** 或者没有开启 **optimizeACK**，也会逐条消息发送 **ACK** 标示；如果目标队列是 Topic 模式，那么无论 **optimizeACK** 是否开启，都会在消费的消息个数  $\geq \text{prefetch} \times 0.5$  时，批量确认这些消息。

## 8. 消费者和生产者性能总结

本小节我们介绍了基于 ActiveMQ 构建的消息队列系统中，生产者和消费者需要关注的重要性能点。但是整个 ActiveMQ 中的性能还需要各位读者在实际工作中，一点一点地去挖掘。这里我们根据已经介绍过的性能关注点进行总结：

- 发送 NON\_PERSISTENT Message 和发送 PERSISTENT Message 是有性能差异的。引起这种差异的原因是前者不需要进行持久化存储；但是这样的性能差异在某些情况下会缩小，例如发送 NON\_PERSISTENT Message 时，由于消费者性能不够导致消息堆积，这时 NON\_PERSISTENT Message 会被转储到物理磁盘上的“temp store”区域。
- 发送带有事务的消息和发送不带有事务的消息，在服务器端的处理性能也是有显著区别的。引起这种差异的原因是带有事务的消息会首先记录在服务器端的“transaction store”区域，并且 ActiveMQ 服务会带有 redo 日志，这样保证发送者端在发送 commit 指令或者 rollback 指令时，服务器会完成相应的处理。
- 在 ActiveMQ 中，为消息生产者所设定的 producerFlowControl 策略非常重要，它确保消息在 ActiveMQ 服务端产生大量堆积的情况下，ActiveMQ 将减缓接收消息，保证了 ActiveMQ 能够稳定工作。你可以通过配置文件设置 producerFlowControl 策略的生效阈值，甚至可以关闭 producerFlowControl 策略（当然不建议这样做）。
- 消息生产者端和消息消费者端都可以通过“异步”方式和服务器进行通信（但是意义不一样）。在生产者端发送异步消息，一定要和 producerWindowSize（回执窗口期）的设置共同使用；在消费者异步接收消息时，要记住有 prefetch 这个关键的预取数值，并且 prefetchSize 在非必要情况下不要设置为 1。很显然适合的 prefetchSize 将改善服务端和消费者端的性能。
- 在 JMS 规范中，消息消费者端也是支持事务的。所谓消费者端的事务是指：一组消息要么全部被 commit（这时消费者会向服务端发送 ACK 标示），要么全部被 rollback（这时同一个消息消费者会向自己重发这些消息，并且这些消息的 redeliveryCounter 属性+1）；进行消息的重发是非常消耗消费者端性能的一件事情，这是因为在这个连接会话中，被 prefetch 但是还没有被处理的消息将一直等待重发的消息最终被确认。
- 为了避免带有错误业务信息的消息被无止境地重发，而影响整个消息系统的性能。在



ActiveMQ 中为超过 `MaximumRedeliveries` 阈值（默认值为 6，但是很明显默认值太高了，建议设置为 3）的消息准备了“死信队列”。

- 只有服务器收到了一条或者一组消息的 `ACK` 标示，才会认为这条或者这组消息被成功处理了。在消费者端有 4 种 `ACK` 工作模式，建议优先选择 `AUTO_ACKNOWLEDGE`。如果这样做了，那么请一定重新改小预取数量、设置 `OptimizeAcknowledge` 为 `true`、重设 `OptimizeAcknowledgeTimeOut` 时间。这样才能保证 `AUTO_ACKNOWLEDGE` 方式工作在“延迟确认”模式下，以便优化 `ACK` 性能。

#### 9.4.4 ActiveMQ 的持久消息存储方案

9.4.3 节曾经讲过，当 ActiveMQ 接收到 `PERSISTENT Message` 消息后就需要借助持久化方案来完成 `PERSISTENT Message` 的存储。这个具体的方案可以是磁盘文件系统，可以是 ActiveMQ 的内置数据库，还可以是某种外部提供的关系型数据库。本节笔者将向读者讲解三种 ActiveMQ 推荐的存储方案的配置使用。

- 如图 9-23 中 2.1 的步骤所示，所有 `PERSISTENT Message` 都要执行持久化存储操作，持久化存储操作方案的性能直接影响着整个 MQ 服务端的 `PERSISTENT Message` 吞吐性能。另外 `NON_PERSISTENT Message` 虽然不会进行持久化存储，但是 `NON_PERSISTENT Message` 也不是永远都只存在于内存区域。
- Topic 模式的工作队列在没有任何活动订阅者的情况下也会对 `PERSISTENT Message` 进行持久化存储吗？当然会，因为 Topic 模式的工作队列还要考虑“`Durable Topic Subscribers`”形式的订阅者。即使没有“`Durable Topic Subscribers`”形式的订阅者，先存储再标记的过程也不会改变（只是不一定真正进入物理磁盘）。
- 另外在客户端启动事务的情况下，如果没有事务的 `commit` 操作，`PERSISTENT Message` 也会进行持久化存储吗？当然还是会的，没有做事务的 `commit` 只是说这些事务中的消息不会进行确认操作，不会分发到某个指定的具体队列中；但是只要使用了 `send` 方法，`PERSISTENT Message` 就会被发送到服务端，就会进行持久化存储操作。
- 如图 9-23 中被标示为 2.2 的操作步骤所示，在 ActiveMQ 设置的持久化方案完成某条消息的持久化后，会在 ActiveMQ 服务节点的内部发出一个“完成”信号。这是为了告诉 ActiveMQ 服务节点，是否可以进行下一步操作。但是为了加快 ActiveMQ 服务节点内部的处理效率，这个过程可以设置为“异步”。
- 进行了持久化存储的 `PERSISTENT Message` 什么时候被删除呢？就如同之前我们提到的一样，ActiveMQ 服务端只有在收到消费者的某一条消息或某一组消息的 `ACK` 标示后，才会认为消息被消费者正确处理了。就是在这个时候，ActiveMQ 会通知持久化方案，进行删除这一条或者这一组消息的操作，并空闲出相应的存储空间。如图 9-23 中



被标示为 5.1 的操作步骤所示。

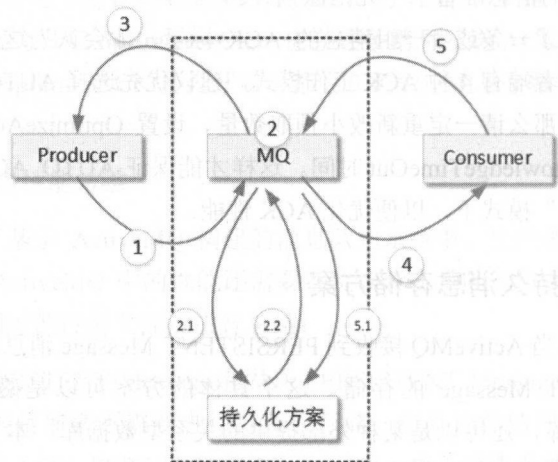


图 9-23 ActiveMQ 中的持久化方案

1. 存储方案配置

在介绍 ActiveMQ 的存储方案之前，首先需要明确的是 ActiveMQ 中的几种“容量”描述：

- **ActiveMQ** 的内核是用 **Java** 编写的，也就是说如果 ActiveMQ 服务没有 Java 环境的支持则无法运行。ActiveMQ 启动时，启动脚本使用 wrapper 包装器来启动 JVM。JVM 相关的配置信息在启动目录的“wrapper.conf”配置文件中。各位读者可以通过改变其中的配置项，设置 JVM 的初始内存大小和最大内存大小（当然还可以进行其他和 JVM 有关的设置，例如开启 debug 模式），如下：

```
.....
# Initial Java Heap Size (in MB)
wrapper.java.initmemory=100
# Maximum Java Heap Size (in MB)
wrapper.java.maxmemory=512
.....
```

以上配置项设置 JVM 的初始内存大小为 100MB，设置 JVM 的最大内存大小为 512MB。如果你在更改后使用 console 参数启动 ActiveMQ，那么会看到当前 ActiveMQ 的 JVM 设置发生了变化，如图 9-24 所示。

```

root@localhost:/usr/apache-activemq-5.13.1/bin
jvm 1 | Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
jvm 1 | Copyright 1999-2006 Tanuki Software, Inc. All Rights Reserved.
jvm 1 |
jvm 1 | Java Runtime: Oracle Corporation 1.7.0_71 /usr/jdk64-7.0.71/jre
jvm 1 | Heap sizes: current=99008k free=92913k max=506816k
jvm 1 | JVM args: -Dactivemq.home=... -Dactivemq.base=... -Djavax.net
jvm 1 | ssl.keyStorePassword=password -Djavax.net.ssl.trustStorePassword=password -Djav
jvm 1 | ax.net.ssl.keyStore=.../conf/broker.ks -Djavax.net.ssl.trustStore=.../conf/b
jvm 1 | roker.ts -Dcom.sun.management.jmxremote -Dorg.apache.activemq.UseDedicatedTaskRu
jvm 1 | nner=true -Djava.util.logging.config.file=logging.properties -Dactivemq.conf=...
jvm 1 | /conf -Dactivemq.data=.../data -Djava.security.auth.login.config=.../conf/
jvm 1 | login.config -Xms100m -Xmx512m -Djava.library.path=.../bin/linux-x86-64/ -Dwra
jvm 1 | pper.key=CuetBdLEkjP0p3pk -Dwrapper.port=32000 -Dwrapper.java.port.min=31000 -Dwr
jvm 1 | apper.java.port.max=31999 -Dwrapper.pid=3821 -Dwrapper.version=3.2.3 -Dwrapper.na
jvm 1 | tive.library=wrapper -Dwrapper.cpu.timeout=10 -Dwrapper.jvmid=1
jvm 1 |
jvm 1 | Extensions classpath:
jvm 1 | [.../lib,.../lib/camel,.../lib/optional,.../lib/web,.../lib
jvm 1 | /extra]
jvm 1 |
jvm 1 | ACTIVEMQ_HOME: ...
jvm 1 | ACTIVEMQ_BASE: ...
jvm 1 | ACTIVEMQ_CONF: .../conf
jvm 1 | ACTIVEMQ_DATA: .../data
jvm 1 | Loading message broker from: xbean:activemq.xml

```

图 9-24 Java 内核要记得调优

- ActiveMQ 每一个服务节点都是一个独立的进程。在 ActiveMQ 主配置文件中，读者可以找到一个“systemUsage”标记，类似定义如下：

```

<systemUsage>
  <systemUsage>
    <memoryUsage>
      <memoryUsage percentOfJvmHeap="70" />
    </memoryUsage>
    <storeUsage>
      <storeUsage limit="100 gb"/>
    </storeUsage>
    <tempUsage>
      <tempUsage limit="50 gb"/>
    </tempUsage>
  </systemUsage>
</systemUsage>

```

- systemUsage: 该标记用于设置整个 ActiveMQ 节点在进程级别的各种“容量”的设置情况。其中可设置的属性包括：①sendFailIfNoSpaceAfterTimeout，当 ActiveMQ 收到一条消息时，如果 ActiveMQ 已经没有多余“容量”了，那么就会等待一段时间（这里设置的毫秒数），如果超过这个等待时间 ActiveMQ 仍然没有可用的容量，那么就拒绝接收这条消息并在消息的发送端抛出 javax.jms.ResourceAllocationException 异常；②sendFailIfNoSpace，当 ActiveMQ 收到一条消息时，如果 ActiveMQ 已经没有多余“容

量”了，就直接拒绝这条消息（不用等待一段时间），并在消息的发送端抛出 `javax.jms.ResourceAllocationException` 异常。

- **memoryUsage**: 该子标记设置整个 ActiveMQ 节点的“可用内存限制”。这个值不能超过刚才你自己设置的 JVM `maxmemory` 的值。其中的 `percentOfJvmHeap` 属性表示使用“百分数值”进行设置，除了这个属性，还可以使用 `limit` 属性进行固定容量授权，例如：`limit="1000MB"`。这些内存容量将供所有队列使用。
- **storeUsage**: 该标记设置整个 ActiveMQ 节点，用于存储“持久化消息”的“可用磁盘空间”。该子标记的 `limit` 属性必须要进行设置。在使用后续介绍的 KahaDB 方案或者 LevelDB 方案进行 PERSISTENT Message 持久化存储时，这个 `storeUsage` 属性就会起作用；如果使用数据库存储方案，那么这个属性就不会起作用了。
- **tempUsage**: 在 ActiveMQ 5.X+ 版本中，一旦 ActiveMQ 服务节点存储的消息达到了 `memoryUsage` 的限制，NON\_PERSISTENT Message 就会被转储到 `temp store` 区域。虽然我们说过 NON\_PERSISTENT Message 不进行持久化存储，但是 ActiveMQ 为了防止“数据洪峰”出现时 NON\_PERSISTENT Message 大量堆积致使内存耗尽的情况出现，还是会将 NON\_PERSISTENT Message 写入到磁盘的临时区域——`temp store`。这个子标记就是为了设置这个 `temp store` 区域的“可用磁盘空间限制”。最后提醒各位读者 `storeUsage` 和 `tempUsage` 并不是“最大可用空间”，而是一个阈值。

## 2. ActiveMQ 中持久化存储方案的演化

说到 ActiveMQ 中持久化存储方案的演化问题，如果仔细阅读 ActiveMQ 官方文档中关于数据持久化技术的描述，就不难发现 ActiveMQ 的开发团队在针对持久化性能问题的优化上可谓与时俱进。这也符合一款健壮软件的生命周期特征：任何功能特性都在进行不断累计完善。

从最初的 AMQ Message Store 方案，到 ActiveMQ V4 版本中推出的 High performance journal（高性能事务支持）附件，并且同步推出了关于关系型数据库的存储方案。ActiveMQ 5.3 版本中又推出了对 KahaDB 的支持（V5.4 版本后称为 ActiveMQ 默认的持久化方案），后来 ActiveMQ V5.8 版本开始支持 LevelDB，而现在的 V5.9+ 版本提供了标准的 ZooKeeper+LevelDB 集群方案。下面我们重点介绍一下 ActiveMQ 中 KahaDB、LevelDB 和关系型数据库这三种持久化存储方案。并且会和读者一起，使用 ZooKeeper 搭建 LevelDB 集群存储方案。

对于最初的 AMQ Message Store 方案，ActiveMQ 官方已不再推荐使用。如果各位读者想了解可以自行搜索相关资料，这里不再进行介绍了。

### 3. KahaDB 存储方案

#### (1) KahaDB 基本结构

KahaDB is a file based persistence database that is local to the message broker that is using it. It has been optimised for fast persistence and is the the default storage mechanism from ActiveMQ 5.4 onwards. KahaDB uses less file descriptors and provides faster recovery than its predecessor, the AMQ Message Store.

以上引用自 Apache ActiveMQ 官方对 KahaDB 的定义。首先 KahaDB 基于文件系统，其次 KahaDB 支持事务。在 ActiveMQ V5.4 版本及后续版本 KahaDB 都是 ActiveMQ 的默认持久化存储方案，Apache ActiveMQ 官方选择它用来替换之前的 AMQ Message Store 存储方案。KahaDB 主要元素包括：一个内存 Metadata Cache 用来在内存中检索消息的存储位置、若干用于记录消息内容的 Data log 文件、一个在磁盘上检索消息存储位置的 Metadata Store，还有一个用于在系统异常关闭后恢复 Btree 结构的 redo 文件。如图 9-25 所示（引用自官方资料）。

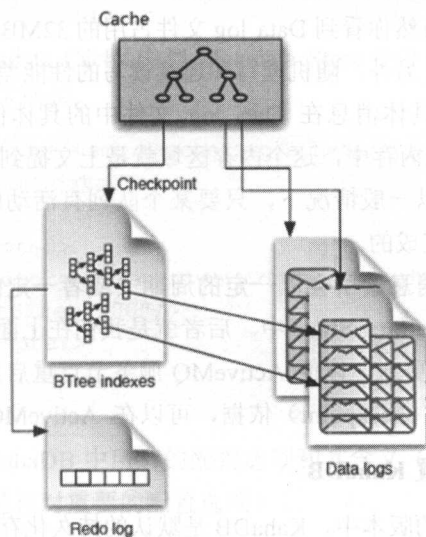


图 9-25 KahaDB 基本结构

以下是 KahaDB 在磁盘文件上的现实展示。注意，可能你查看自己测试实例中所运行的 KahaDB，看到的效果和本书中给出的效果不完全一致。例如 Data log 文件可能叫 db-1.log，也有可能多出个 db.free 的文件，但是这些都不影响我们对文件结构的分析：

.....

```
[root@localhost KahaDB]# ll -h
```

总用量 29MB

```
-rw-r--r--. 1 root root 32M 4月 7 04:53 db-3.log
-rw-r--r--. 1 root root 7.6M 4月 7 04:53 db.data
-rw-r--r--. 1 root root 2.8M 4月 7 04:53 db.redo
-rw-r--r--. 1 root root 8 4月 7 04:50 lock
.....
```

- **db-3.log**: 这个文件就是我们上面提到的 Data log 文件。一个 KahaDB 中, 可能同时存在多个 Data log 文件, 它们存储了每一条持久化消息的真正内容。这些 Data log 文件统一采用 db-\*.log 的格式进行命名, 并且每个 Data log 文件默认的大小都是 32MB (当然可以进行调整)。当一个 Data log 文件中的所有消息被成功消费后, 这个 Data log 文件会在 Metadata Cache 中被标记为删除, 并在下个 checkpoint 周期进行删除操作。
- 各位读者可能已经注意到一个现象: 为什么 db-3.log 的默认占用大小就是 32MB, 但是目录显示的“总用量”却只有 29MB 呢? 在这个文件夹中, 除了 db-3.log 文件本身, 加上其他几个文件所占用的大小, 已经远远超过了 32MB! 这是因为, 为了加快写文件的性能, Data log 文件采用顺序写的方式进行操作, 为了保证文件使用的扇区在物理上是连续的, 所以 Data log 文件会预占这些扇区 (这个和 Hadoop 中每一个 block 大小都是固定的原因类似)。虽然你看到 Data log 文件占用的 32MB 的磁盘空间, 但是这些磁盘空间并没有全部使用。另外, 随机读写和连续读写的性能差异是非常巨大的。
- 为了更快地找到某个具体消息在 Data log 文件中的具体位置。消息的位置索引采用 BTree 的结构被存储在内存中, 这个内存区域就是上文提到的 Metadata Cache (大小也是可以设置的)。所以一般情况下, 只要某个队列有活动的消费者存在, 消息的定位、读取操作是可以很快完成的。
- 内存中没有被处理的消息索引会以一定的周期 (或者一定的数量规模) 为依据, 同步 (checkpoint) 到 Metadata Store 中。后者就是我们在上面看到的“db.data”文件, 当然 db.redo 文件也会被更新, 以便 ActiveMQ 服务节点重启后对 Metadata Cache 进行恢复。最后, 消息同步 (checkpoint) 依据, 可以在 ActiveMQ 的主配置文件中设置。

## (2) 在 ActiveMQ 中配置 KahaDB

由于在 ActiveMQ V5.4+ 的版本中, KahaDB 是默认的持久化存储方案。所以即使你不配置任何的 KahaDB 参数信息, ActiveMQ 也会启动 KahaDB。这种情况下, KahaDB 文件所在位置是你的 ActiveMQ 安装路径下的/data/\${broker.Name}/KahaDB 子目录。其中 \${broker.Name} 代表这个 ActiveMQ 服务节点的名称。

在正式的生产环境中, 还是建议你在主配置文件中明确设置 KahaDB 的工作参数。如下所示:

```
<broker xmlns=http://activemq.apache.org/schema/core brokerName="localhost"
dataDirectory="${activemq.data}">
```



```

.....
<persistenceAdapter>
  <kahaDB directory="${activemq.data}/kahadb"/>
</persistenceAdapter>
.....
</broker>

```

以上配置项设置使用 kahaDB 为持久化存储方法，并且设置 kahaDB 的工作目录为 ActiveMQ 安装路径下的/data/kahadb 目录。如果你需要调整 Data log 文件默认的 32MB 的大小，可以使用 journalMaxFileLength 属性进行设置，如下所示：

```

<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost"
dataDirectory="${activemq.data}">
.....
  <persistenceAdapter>
    <kahaDB directory="${activemq.data}/kahadb" journalMaxFileLength=
"64mb"/>
  </persistenceAdapter>
.....
</broker>

```

还可以设置为：当 Metadata Cache 中和 Metadata Store 中不同的索引条数达到 500 条时，就进行 checkpoint 同步。如下所示：

```

<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost"
dataDirectory="${activemq.data}">
.....
  <persistenceAdapter>
    <kahaDB directory="${activemq.data}/kahadb" journalMaxFileLength
="64mb" indexWriteBatchSize="500"/>
  </persistenceAdapter>
.....
</broker>

```

表 9-2 为读者示例了 KahaDB 中所有的配置选项和其含义（表 9-2 基于 ActiveMQ 官方资料进行整理，加“\*”部分是相对重要的配置选项）。

表 9-2

property name	default value	Comments
*directory	activemq-data	消息文件和日志的存储目录
*indexWriteBatchSize	1000	当 Metadata cache 区域和 Metadata store 区域不同的索引数量达到这个值后，Metadata cache 将会发起 checkpoint 同步

续表

property name	default value	Comments
*indexCacheSize	10000	内存中，索引的页大小。超过这个大小 Metadata cache 将会发起 checkpoint 同步
*enableIndexWriteAsync	false	索引是否异步写到消息文件中，所以不要设置为 true
*journalMaxFileLength	32mb	一个消息文件的大小
*enableJournalDiskSyncs	true	如果为 true，则保证使用同步写入的方式持久化消息到 journal 文件中
*cleanupInterval	30000	清除（清除或归档）不再使用的 db-*.log 文件的时间周期（毫秒）
*checkpointInterval	5000	写入索引信息到 metadata store 中的时间周期（毫秒）
ignoreMissingJournalfiles	false	是否忽略丢失的 journal 文件。如果为 false，当丢失了 journal 文件时，那么 broker 启动时会抛异常并关闭
checkForCorruptJournalFiles	false	检查消息文件是否损坏，true，检查发现损坏会尝试修复
checksumJournalFiles	false	产生一个 checksum，以便能够检测 journal 文件是否损坏
property name	default value	Comments

- 5.4 版本之后有效的属性，如表 9-3 所示。

表 9-3

property name	default value	Comments
*archiveDataLogs	false	当为 true 时，归档的消息文件被移到 directoryArchive，而不是直接删除
*directoryArchive	null	存储被归档的消息文件目录
databaseLockedWaitDelay	10000	在使用负载时，等待获得文件锁的延迟时间，单位为毫秒
maxAsyncJobs	10000	等待写入 journal 文件的任务队列的最大数量。应该大于或等于最大并发 Producer 的数量。配合并行存储转发属性使用
concurrentStoreAndDispatchTopics	false	如果为 true，则转发消息的时候同时提交事务
concurrentStoreAndDispatchQueues	true	如果为 true，则转发 Topic 消息的时候同时存储消息在 message store 中

- 5.6 版本之后有效的属性，如表 9-4 所示。

表 9-4

property name	default value	Comments
archiveCorruptedIndex	false	是否归档错误的索引到 Archive 文件夹下

- 5.10 版本之后有效的属性，如表 9-5 所示。

表 9-5

property name	default value	Comments
IndexDirectory		单独设置 KahaDB 中 db.data 文件的存储位置。如果不进行设置，则 db.data 文件的存储位置还是将以 directory 属性设置的值为准

#### 4. LevelDB 存储方案

LevelDB 是能够处理十亿级别规模的，基于 Key-Value 型数据结构的 C++ 程序库，由 Google 发起并开源。LevelDB 只能由本操作系统的其他进程调用，所以它不具有网络性。如果需要网络上的远程进程操作 LevelDB，那么就要自行封装服务层。

##### (1) LevelDB 基本结构

LevelDB 中的核心设计算法是跳跃表（Skip List），核心操作策略是对磁盘上的数据日志结构进行归并（LSM）。跳跃表实际上是二叉平衡树的一种变形结构，它通过将一个有序链表进行“升维”操作，从而减少每一层上需要遍历的数据数量，达到快速查找的目的。图 9-26 示意了一个跳跃表结构（在实际工作中，跳跃表的层级和“升维”策略的不同，跳跃表的结构也不一样）。

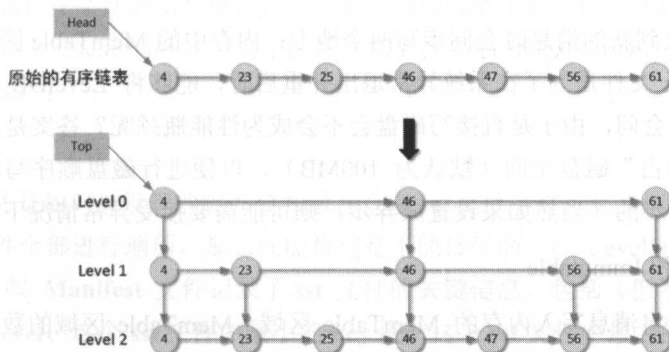


图 9-26 跳跃表结构

可以将图 9-26 中的每个元素节点，想象成每一条消息的 key 值。为了讲解方便，上图中笔者将拥有全部数据的元素的跳跃层称为 Level 2（最高层），但实际上规范的跳跃表结构中，拥有全部元素的层次称为 Level 0（最底层）。跳跃表的结构并非一成不变，当有一条新的记录需要插入到结构时，可能会引起表中的多个 Level 都发生变化。那么 LevelDB 是如何应用跳跃表结构的？又是如何进行归并操作的？我们首先来看看 LevelDB 的简要结构（图 9-27）。

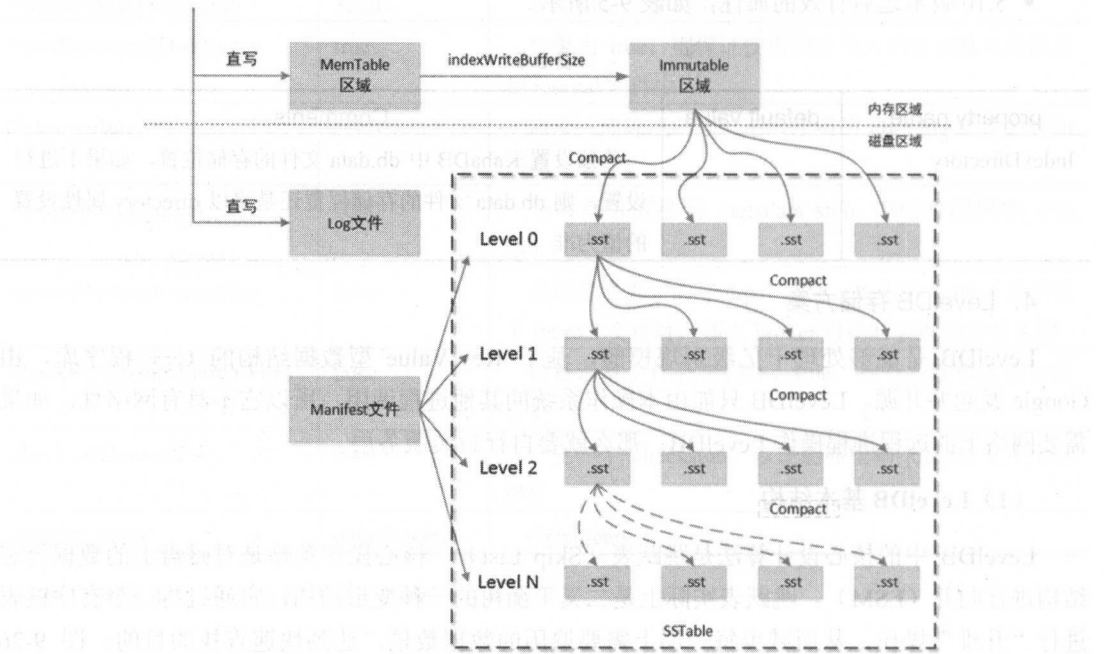


图 9-27 LevelDB 数据结构

• Log 文件

当 LevelDB 收到新的消息时会同步写两个地方：内存中的 MemTable 区域和磁盘上的 Log 文件。直接写 Log 文件是为了在系统异常退出并重启时，能够将 LevelDB 恢复到退出前的结构；那么有的读者会问，由于是直接写磁盘会不会成为性能瓶颈呢？答案是，LevelDB 的 Log 文件操作需要“预占”磁盘空间（默认为 100MB），以便进行磁盘顺序写的操作。并且这个过程可以设置为异步的（当然如果设置成异步，则可能需要接受异常情况下数据丢失的风险）。

• MemTable 和 Immutable

LevelDB 还会将消息写入内存的 MemTable 区域，MemTable 区域的数据组织结构就是跳跃表（Skip List），这样的数据组织结构可以在读取内存中信息的时候，快速完成信息定位。当 MemTable 区域的数据量达到 indexWriteBufferSize 属性设置的大小时（默认为 6MB），

LevelDB 就会把这个 MemTable 区域标记为 Immutable，并开启一个新的 MemTable 区域。一定注意，是标记为 Immutable，而不是把 MemTable 区域的数据复制到某一个 Immutable 区域。

新标记的 Immutable 区域中的数据会被执行 Compact 操作，从而写入到磁盘上的.sst 文件中。所谓 Compact 操作是指：LevelDB 会剔除 Immutable 区域中那些已经被标示为“删除”的数据（成功消费的数据就会被标记为“删除”），排除那些格式错误的的数据，并可能进行数据压缩。

#### • SSTable 文件

SSTable 文件是指存在于硬盘上，后缀名为.sst 的文件。这些文件是 LevelDB 磁盘上最重要的数据记录文件，每一个 SSTable 文件的默认大小为 2MB，也就是说 LevelDB 的文件夹下会有很多的.sst 文件。SSTable 文件并不是顺序写的，而是按照数据的 key 排序进行随机写，所以 SSTable 文件无须“预占”存储磁盘存储空间。

借鉴于跳跃表的设计思想，SSTable 文件也是分层次的。每一层可存储的数据量是上一层的 10 倍。举个例子，第 Level 2 层可存储的数据量 80MB，那么第 Level 3 层可存储的数据量就是 800MB。当某一层可存储的数据量达到最大值时，LevelDB 就会在“当层”选取一个.sst 文件，向下层做 Compact 操作，由于来自于上层的新数据，所以下层的.sst 文件内容将产生变化（上文说过，.sst 文件中的内容是按照数据的 key 排序的）。

每一个 SSTable 文件，由多个 Block 块构成（默认大小为 4KB），Block 块是 LevelDB 读写磁盘上 SSTable 文件的最小单元。每一个 SSTable 文件最后一个 Block 块称为 Index Block，它指明了 SSTable 文件中每一个 Data Block 的起始位置。

但是每次读取某个 Block 块时，如果都在磁盘上先去找 Index Block，然后再根据其中记录的 index，找到 Block 在文件的起始位置，那么查找效率显然不高。所以 LevelDB 的内存区域中，有一个称为 Block Cache 的区域。这个区域存储着众多的 Index Block，这样就不需要到磁盘上查找 Index Block 了。

#### • Manifest

众多的.sst 文件是如何被管理的呢？要知道在众多.sst 文件中进行某条消息的查找时，如果将某一层的.sst 文件全部进行遍历，那么性能肯定是不能接受的。在 LevelDB 中有一类文件被称为 Manifest，这些 Manifest 文件记录了.sst 文件的关键信息，包括（但不限于）：某个.sst 文件属于哪一个 Level、这个.sst 文件中最小的 key 值、这个.sst 文件中最大的 key 值。

### （2）在 ActiveMQ 中配置 LevelDB



在 ActiveMQ 中配置使用 LevelDB 作为持久化存储方案实际上很简单,使用主配置文件中的 `persistenceAdapter` 标记就可以完成。最简配置如下所示:

```
.....
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost"
dataDirectory="${activemq.data}">
.....
<persistenceAdapter>
  <levelDB directory="${activemq.data}/levelDB"/>
</persistenceAdapter>
.....
</broker>
.....
```

以上示例配置中, `directory` 属性表示 LevelDB 的结构文件所放置的目录位置。请注意,由于 Log 文件是顺序写的机制,所以 Log 文件也会“预占”磁盘空间,并且 Log 文件默认的大小就是 **100MB**。那么只要生成一个 Log 文件,就至少会占据 100MB 的存储空间。也就是说,如果将主配置文件中 `storeUsage` 标记的 `limit` 属性设置为 200MB,那么透过 ActiveMQ 管理界面看到的现象就是:只要有任何一条 PERSISTENT Message 形式的消息被接收,Store percent used 立刻就会变成 50%。如果将 `storeUsage` 标记的 `limit` 属性设置为 100MB,那么只要有任何一条 PERSISTENT Message 形式的消息被接收,ActiveMQ 服务端的 Producer Flow Control 策略就会立刻开始工作。

所以一定不要吝啬分配 `memoryUsage`、`storeUsage`。依据你的团队在生产环境下的存储方案,也可以通过 `logSize` 属性改变 LevelDB 中单个 Log 文件的大小。如下示例:

```
<!-- 限制成 50MB -->
<persistenceAdapter>
  <levelDB directory="${activemq.data}/levelDB" logSize="52428800"/>
</persistenceAdapter>
```

在默认的 LevelDB 存储策略中,当 ActiveMQ 接收到一条消息后,就会同步将这条消息写入到 Log 文件中,并且同时在内存区域向 Memtable 写入位置索引。通过配置你也可以将这个过程改为“异步”:

```
<!-- 改为异步写 Log 文件 -->
<persistenceAdapter>
  <levelDB directory="${activemq.data}/levelDB" logSize="52428800" sync=
"false"/>
</persistenceAdapter>
```

表 9-6 展示了可以使用的 LevelDB 的配置属性,使用“\*”标识出来的属性是笔者认为相对重要的配置属性。

表 9-6

property name	default value	Comments
*directory	“LevelDB”	数据文件的存储目录
sync	true	是否进行磁盘的同步写操作
*logSize	104857600 (100 MB)	Log 日志文件的最大值
verifyChecksums	false	是否对从文件系统中读取的数据进行强制校验
paranoidChecks	false	如果 LevelDB 检测到数据错误，则尽快将错误在存储位置进行标记
indexFactory	org.fusesource.leveldbjni.JniDBFactory, org.iq80.leveldb.impl.Iq80DBFactory	创建 LevelDB 时使用的工厂类，由于 LevelDB 的本质是 C++ 程序库，所以 Java 是通过 Jni 进行底层调用的
*indexMaxOpenFiles	1000	可供索引使用的打开文件的数量，这是因为 Level 内部使用了多线程进行文件读写操作
indexWriteBufferSize	6291456 (6 MB)	内存 MemTable 的最大值，如果 MemTable 达到这个值，就会被标记为 Immutable
indexBlockSize	4096 (4 K)	每个读取到内存的 SSTable——Index Block 数据的大小
*indexCacheSize	268435456 (256 MB)	使用一个内存区域记录多个 Level 中，SSTable——Index Block 数据，以便读操作时，不经过遍历就可直接定位数据在某个 Level 中的位置，建议增大该区域
indexCompression	snappy	适用于索引块的压缩类型，影响 Compression 策略
logCompression	none	适用于日志记录的压缩类型，影响 Compression 策略
property name	default value	Comments

5. 关系型数据库存储方案

从 ActiveMQ V4 版本开始，支持使用关系型数据库进行持久化存储——通过 JDBC 实现的数据库连接。可以支持的关系型数据库包括（但不限于）：Apache Derby、DB2、HSQL、Informix、MySQL、Oracle、Postgresql、SQLServer、Sybase。

下面向各位读者演示如何为 ActiveMQ 配置 MySQL 数据库服务。前提是已经在某个网络位置准备好了 MySQL 服务，并可以成功进行远程登录。

```
.....
<broker>
.....
<!-- 配置 ActiveMQ 连接到 MySQL 服务 -->
```

```

<!-- 记得删除原来的 KahaDB 或者 LevelDB 的配置 -->
<persistenceAdapter>
  <jdbcPersistenceAdapter dataSource="#mysql_datasource" create
Tables OnStartup="true"/>
</persistenceAdapter>
.....
</broker>
<!-- 演示使用的是 C3P0 连接池，当然也可以使用 DBCP 连接池 -->
<!-- 就是 Spring 的配置文件结构 -->
<bean id="mysql_datasource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="driverClass" value="com.mysql.jdbc.Driver"/>
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/activemqdb?relaxAutoCommit=true&useUnicode=true&characterEncoding=utf-8"/>
  <property name="user" value="root"/>
  <property name="password" value="123456"/>
  <property name="minPoolSize" value="10"/>
  <property name="maxPoolSize" value="30"/>
  <property name="initialPoolSize" value="10"/>
</bean>
.....

```

在配置关系型数据库作为 ActiveMQ 的持久化存储方案时，要注意几个事项：

- 配置信息建议放置在你的 jetty.xml 配置文件中，也可以放置在 activemq.xml 配置文件中。除此之外，还要记得需要使用到的相关 jar 文件放置到 ActiveMQ 安装路径下的 /lib 目录。例如使用 MySQL + C3P0 的配置中，需要的 jar 包至少包括：mysql-jdbc 驱动的 jar 包和 C3P0 的 jar 包。
- 在 jdbcPersistenceAdapter 标签中，我们设置了 createTablesOnStartup 属性为 true，这是为了在第一次启动 ActiveMQ 时，ActiveMQ 服务节点会自动创建所需要的数据表。启动完成后，可以去掉这个属性，或者更改 createTablesOnStartup 属性为 false。
- 在配置和测试的过程中，你可能会遇到这样的问题：“Java.lang.IllegalStateException: BeanFactory not initialized or already closed”。这是因为你的操作系统的机器名中有 “\_” 符号。更改机器名并且重启后，即可解决问题。

在同样的硬件资源条件下，相比 KahaDB 和 LevelDB 这样“内存+存储介质”的持久化方案而言，使用关系型数据库作为 ActiveMQ 的持久化方案绝对不能说性能最好，但是在大多数情况下这个持久化方案也不会成为整个顶层架构的设计瓶颈（因为关系型数据库一般都有高可用和高负载方案）。所以很多团队还是会使用这样的持久化方案，很大一部分原因就是这些团队对关系型数据库有更丰富的使用经验，且有专门的数据库管理人员。

## 6. 如何选用持久化存储方案

以上我们介绍了三种持久化存储方案：KahaDB、LevelDB、关系型数据库。其中 KahaDB 和 LevelDB 的工作原理基本类似，都采用内存+磁盘介质的方案：内存用于存放信息的位置索引，磁盘介质上存放消息内容。而关系型数据库的方案，ActiveMQ 将完全通过 JDBC 对数据库进行操作完成消息的存储和修改。

但是并不是如网络上一些资料所说的那样，一定要对三种持久化存储方案的速度做比较后，选择最快的那种存储方案。这里面至少有两个误区：

(1) 某种存储方案的速度一定比另一种存储方案的速度快。

(2) 一定要选择速度快的那种存储方案。

下面我们进行一些讨论：

- 根据不同的硬件层配置，同一种持久化存储方案的性能是完全不一样的。例如在单节点计算的情况下，选用 DDR 2133 双通道内存组和 DDR3 1333 单通道内存条从理论上至少就可以多获得 4Gbps 的带宽；选用同样支持 SATA3 规范的机械硬盘和 SSD 固态硬盘，虽然两者理论上的对外速度都标称 6Gbps，但是由于机械硬盘上单磁头的读写速度存在瓶颈，所以就算进行连续读操作，速度也只能达到 200MB/s 左右。但是固态硬盘的连续读速度却可以达到 500MB/s 左右（基本已经接近 6Gbps 的理论峰值）。
- 如果是企业级硬件存储方案，那么速度差异还会继续扩大。例如电信行业经常采用的 IBM 各个系列磁盘阵列，一般都会配置诸如 RAID5 这样的软存储方案。这样一来，同一份文件有多个副本，并且有多个磁头负责读写。磁盘阵列的对外输出一般会采用光纤通道（FC），而光纤通道行业协会（Fibre Channel Industry Association）最新推出的（2015 年实施）Gen 6 第 6 代光纤通道标准中，设计的对外传输理论速度是 128Gbps。
- 当然，除非你的公司/团队能够接受这些企业级存储方案高昂的费用。否则还是建议在生产环境搭建性价比较高的折中方案。例如采用 20 台左右 PC Server 搭建 Ceph/MFS 分布式存储系统。

某种存储方案的性能，除了考虑这种存储方案的工作原理对其有直接影响，还要考虑它的工作环境。只有根据软件团队预估的系统压力、综合建设方案、考虑后续扩容方式，来确定采用哪一种存储方案，才是科学的。

## 9.5 MQ 实践：ActiveMQ 集群方案

在生产环境中为了保证让我们设计的 MQ 方案能够持续工作，还需要为它搭建集群环境，

从而保证 MQ 服务的可靠性和它的处理性能。集群方案主要为了解决系统架构中的两个关键问题：高可用和高性能。

ActiveMQ 服务的高可用性是指，在 ActiveMQ 服务性能不变、数据不丢失的前提下，确保当系统灾难出现时 ActiveMQ 能够持续提供消息服务，高可靠性方案的最终目的是减少整个 ActiveMQ 停止服务的时间。ActiveMQ 服务的高性能是指，在保证 ActiveMQ 服务持续稳定性、数据不丢失的前提下，确保 ActiveMQ 集群能够在单位时间内吞吐更高数量的消息、确保 ActiveMQ 集群处理单条消息的时间更短、确保 ActiveMQ 集群能够容纳更多的客户端稳定连接。下面我们分别介绍如何通过多个 ActiveMQ 服务节点集群方式，分别提供高可用方案和高性能方案。

### 9.5.1 ActiveMQ 高性能方案

ActiveMQ 的多节点集群方案，主要有动态集群和静态集群两种方案。所谓动态集群就是指，同时提供消息服务的 ActiveMQ 节点数量、位置（IP 和端口）是不确定的，当某一个节点启动后，会通过网络组播的方式向其他节点发送通知（同时接收其他节点的组播信息）。网络中其他节点收到组播通知后，就会向这个节点发起连接，最终将新的节点加入 ActiveMQ 集群（图 9-28）；静态集群是指同时提供消息服务的多个节点的位置（IP 和端口）是确定的，每个节点不需要通过广播的方式发现集群中的其他节点，只需要在启动时按照给定的位置进行连接（图 9-29）。

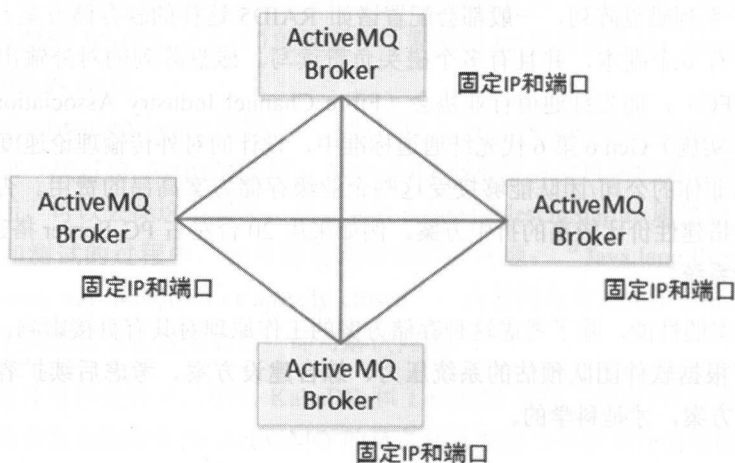


图 9-28 静态集群方案



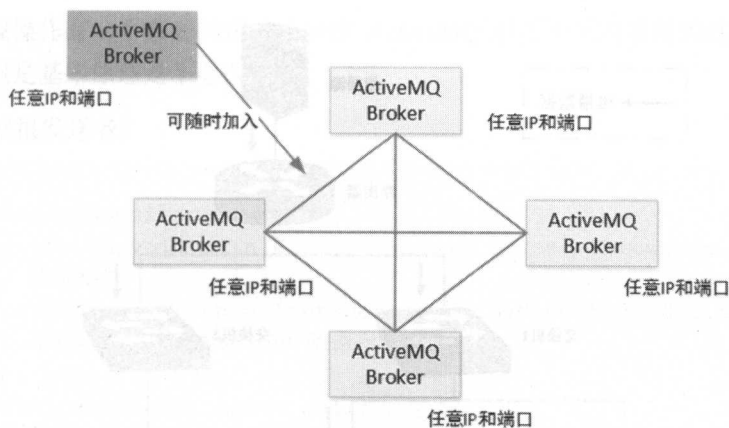


图 9-29 动态集群方案

### 1. 基于组播（multicast）的节点发现

在使用动态集群配置时，当某个 ActiveMQ 服务节点启动后并不知道整个网络中还存在哪些其他的服务节点。所以 ActiveMQ 集群需要使用一种节点与节点间的发现机制，以保证能够解决上述问题。目前在分布式系统常用的节点主动发现方式包含三种，它们是：

- 协调器/注册器发现方式：这种方式一般使用类似 ZooKeeper、Spring Cloud Eureka 这样的分布协调器或者注册中心进行节点/服务注册，以便新节点上线后能够通知/查询到其他已上线的节点信息。
- 结构化和非结构化的 P2P 节点/网络发现方式：这其实是一系列节点发现算法的统称，常用于跨 NAT 设备的庞大规模集群——是的，我们正在讨论目前 P2P 技术领域流行的节点发现技术。这个领域并不是本书所计划涉及的领域，但为了感兴趣的读者能够有些学习思路，这里向读者列举一些 P2P 技术领域常用的节点发现算法，例如基于 DHT 的搜索算法、Chord 网络搜索算法、Flooding 搜索算法和随机漫步算法。最后，再多说一句，以上提到的协调器/注册器方式实际上在第一代 P2P 技术中也有应用，被称为集中式网络搜索算法。
- “组播”发现方式：ActiveMQ 集群的动态主要使用“组播”方式进行其他节点的发现。所以本书会针对这种节点发现方式多介绍几句。

**组播（multicast）**基于 UDP 协议，它是指在一个可连通的网络中，某一个数据报发送源向一组数据报接收目标进行操作的过程。在这个过程中，数据报发送者只需要向这个组播地址（一个 D 类 IP）发送一个数据报，那么加入这个组播地址的所有接收者都可以收到这个数据报。组播实现了网络中单点到多点的高效数据传送，能够节约大量网络带宽，降低网络负载（图 9-30）。

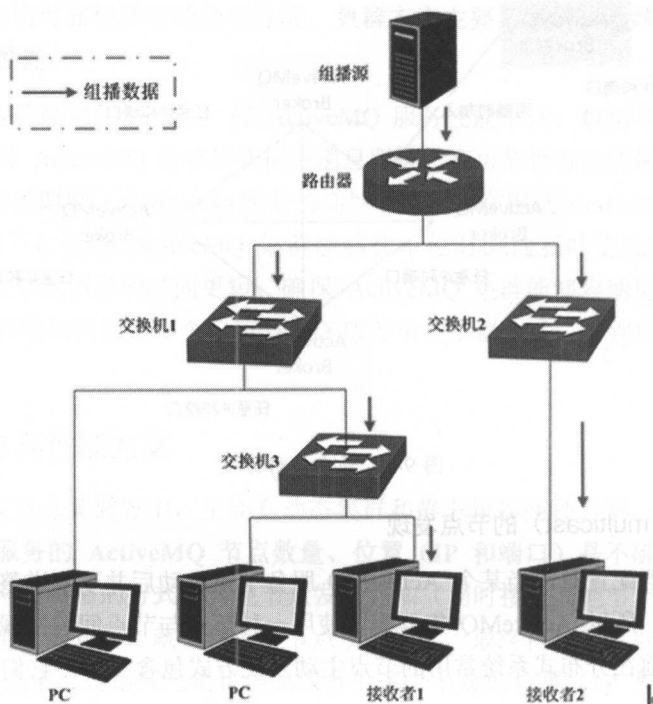


图 9-30 组播示意图

在 IP 协议中，规定的 D 类 IP 地址为组播地址。224.0.0.0~239.255.255.255 这个范围内的 IP 都是 D 类 IP 地址，其中有一些 IP 段是保留的有特殊含义的。

- 224.0.0.0~224.0.0.255：这个 D 类 IP 地址段为保留地址，不建议在开发过程中使用，因为可能产生冲突。例如 224.0.0.5 这个组播地址专供 OSPF 协议（是一种路由策略协议，用于找到最优路径）使用的组播地址；224.0.0.18 这个组播地址专供 VRRP 协议使用（VRRP 协议是虚拟路由器冗余协议）。
- 224.0.1.0~224.0.1.255：这个 D 类 IP 地址为公用组播地址，用于在整个 Internet 网络上进行组播。除非有顶级 DNS 的控制/改写权限，否则不建议在局域网内使用这个组播地址段。
- 239.0.0.0~239.255.255.255：这个 D 类 IP 地址段为推荐在局域网内使用的组播地址段。注意，如果要在局域网内使用组播功能，需要局域网中的交换机/路由器支持组播功能。幸运的是，目前市场上只要不是太过低端的交换机/路由器，都支持组播功能（组播功能所使用的主要协议为 IGMP 协议）。

下面使用 Java 语言，编写一个局域网内的组播发送和接收过程。以便让各位读者对基于

组播的节点发现操作有一个直观的理解。虽然 ActiveMQ 中关于节点发现的过程要比以下的示例复杂得多，但是基本原理是不会改变的。

- 组播数据报发送者：

```
.....
public class SendMulticast {
    public static void main(String[] args) throws Throwable {
        // 组播地址
        InetAddress group = InetAddress.getByName("239.0.0.5");
        // 组播端口，同时也是 UDP 数据报的发送端口
        int port = 19999;
        MulticastSocket mss = null;
        // 创建一个用于发送/接收的 MulticastSocket 组播套接字对象
        mss = new MulticastSocket(port);
        // 创建要发送的组播信息和 UDP 数据报
        // 携带的数据内容，就是这个 ActiveMQ 服务节点用来提供 Network Connectors
        // 的 TCP/IP 地址和端口等信息
        String message = "我是一个活动的 ActiveMQ 服务节点（节点编号:yyyyyyyy），
我的可用 tcp 信息为: XXXXXXXXXX : ";
        byte[] buffer2 = message.getBytes();
        DatagramPacket dp = new DatagramPacket(buffer2, buffer2.length,
group, port);
        // 使用组播套接字 joinGroup(), 将其加入到一个组播
        mss.joinGroup(group);
        // 开始按照一定的周期向加入到 224.0.0.5 组播地址的其他 ActiveMQ 服务节点进
        // 行广播
        Thread thread = Thread.currentThread();
        while (!thread.isInterrupted()) {
            // 使用组播套接字的 send() 方法，将组播数据包对象放入其中，发送组播数据包
            mss.send(dp);
            System.out.println(new Date() + "发起组播: " + message);
            synchronized (SendMulticast.class) {
                SendMulticast.class.wait(5000);
            }
        }
        mss.close();
    }
}
```

- 组播数据报接收者：

```
.....
//测试接收组播信息
public class AcceptMulticast {
```

```
public static void main(String[] args) throws Throwable {
    // 建立组播套接字, 并加入分组
    MulticastSocket multicastSocket = new MulticastSocket(19999);
    // 注意, 组播地址和端口必须和发送者的一致, 才能加入正确的组
    InetAddress ad = InetAddress.getByName("239.0.0.5");
    multicastSocket.joinGroup(ad);
    // 准备接收可能的组播信号
    byte[] datas = new byte[2048];
    DatagramPacket data = new DatagramPacket(datas, 2048, ad, 19999);
    Thread thread = Thread.currentThread();
    // 开始接收组播信息, 并打印出来
    System.out.println(".....开始接收组播信息.....");
    while(!thread.isInterrupted()) {
        multicastSocket.receive(data);
        int leng = data.getLength();
        System.out.println(new String(data.getData(), 0, leng, "UTF-8"));
    }
    multicastSocket.close();
}
```

## 2. 桥接 Network Bridges

为了实现 ActiveMQ 集群的横向扩展要求和高稳定性要求, ActiveMQ 集群提供了 Network Bridges 功能。通过 Network Bridges 功能, 技术人员可以将多个 ActiveMQ 服务节点连接起来。并让它们通过配置好的策略作为一个整体对外提供服务。这样的服务策略主要包括两种: 主/从模式和负载均衡模式(图 9-31)。

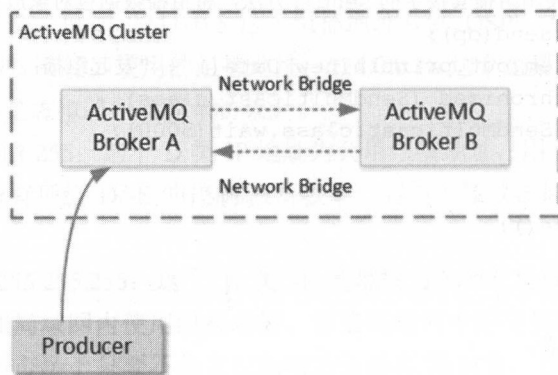


图 9-31 桥接示意

### 3. 动态 Network Connectors

既然已经讲述了 ActiveMQ 中的动态节点发现原理和 ActiveMQ Network Bridges 的概念，那么关于 ActiveMQ 怎样配置集群的方式就是非常简单的问题了。我们先来讨论如何进行基于组播发现的 ActiveMQ 负载均衡模式的配置——动态网络连接 Network Connectors；再来讨论基于固定地址的负载均衡模式配置——静态网络连接 Network Connectors。

要配置基于组播发现的 ActiveMQ 负载均衡模式，开发人员只需要在每一个 ActiveMQ 服务节点的主配置文件中（activemq.xml），添加/更改类似如下配置信息即可：

```
<transportConnectors>
  <!-- 在 transportConnector 中增加 discoveryUri 属性，表示这个
transportConnector 是要通过组播告知其他节点的：使用这个 transportConnector 位置连接我
-->
  <transportConnector name="auto" uri="auto+nio://0.0.0.0:61616?
maximumConnections=1000&wireFormat.maxFrameSize=104857600&org.apache.
activemq.transport.nio.SelectorManager.corePoolSize=20&org.apache.active
mq.transport.nio.SelectorManager.maximumPoolSize=50&consumer.prefetchSiz
e=5" discoveryUri="multicast://239.0.0.5" />
</transportConnectors>
.....
<!-- 关键的 networkConnector 标签，URI 属性标示为组播发现-->
<networkConnectors>
  <networkConnector uri="multicast://239.0.0.5" duplex="false"/>
</networkConnectors>
```

#### (1) networkConnector 标签

如果使用 ActiveMQ 的组播发现功能，那么请在 networkConnector 标签的 uri 属性中添加如下格式的信息：

```
multicast://[组播地址][:端口]
```

例如，可以按照如下方式使用 ActiveMQ 默认的组播地址来发现网络种其他 ActiveMQ 服务节点：

```
#ActiveMQ 集群默认的组播地址 (239.255.2.3) :
multicast://default
```

也可以按照如下方式，指定一个组播地址——这在高安全级别的网络中很有用，因为可能其他的组播地址已经被管理员禁用。注意组播地址只能是 D 类 IP 地址段：

```
#使用组播地址 239.0.0.5
multicast://239.0.0.5
```

如图 9-32 所示是通过抓包软件获得的的组播 UDP 数据报文。



Time	Source	Destination	Protocol	Info
2 7696.631993	192.168.61.139	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
3 7696.632588	192.168.61.138	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
4 7697.134032	192.168.61.138	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
5 7697.134529	192.168.61.139	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
6 7697.635479	192.168.61.139	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
7 7697.635480	192.168.61.138	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
8 7698.137398	192.168.61.139	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
9 7698.137399	192.168.61.138	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
0 7698.639281	192.168.61.139	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
1 7698.639282	192.168.61.138	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
2 7699.140225	192.168.61.138	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
3 7699.140659	192.168.61.139	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
4 7699.646217	192.168.61.138	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
5 7699.646796	192.168.61.139	239.0.0.5	UDP	Source port: 6155 Destination port: 6155
Frame 31272: 103 bytes on wire (824 bits), 103 bytes captured (824 bits)				
Ethernet II, Src: Vmware_01:02:1f (00:0c:29:01:02:1f), Dst: IPv4mcast_00:00:05 (01:00:5e:00:00:05)				
Internet Protocol, Src: 192.168.61.138 (192.168.61.138), Dst: 239.0.0.5 (239.0.0.5)				
User Datagram Protocol, Src Port: 6155 (6155), Dst Port: 6155 (6155)				
Data (61 bytes)				
Data: 64696661756c742e4163746976654d512d342e616c697665...				
[Length: 61]				
0000	01 00 5e 00 00 05 00 0c	29 01 02 1f 03 00 45 00	.....E	
0010	00 39 00 00 40 00 01 11	8c 5c c0 a8 3d 8a ef 00	.Y. @... ..	
0020	00 05 18 0b 18 0b 00 45	0d 8d 84 65 66 61 73 6c	.....E default	
0030	74 2e 41 63 74 69 76 65	4d 31 2d 34 2e 61 6c 69	t.ActiveMQ-4.all	
0040	76 65 2e 25 6c 6f 63 61	6c 68 6f 73 74 25 61 75	ve,localhost	
0050	74 6f 2b 6e 69 6f 3a 2f	2f 61 63 74 69 76 65 6a	to:nio: /activem	
0060	74 3a 36 31 36 31 36 31	36 31 36 31 36 31 36 31	616161	

图 9-32 组播 UDP 抓包信息

从图 9-32 中我们可以获得几个关键信息：

- 192.168.61.138 和 192.168.61.139 这两个 IP 地址分别按照一定的周期（1 秒一次），向组播地址 239.0.0.5 发送 UDP 数据报。以便让在这个组播地址的其他服务节点能够感知自己的存在。
- 以上 UDP 数据报文使用的端口是 6155。也可以更改这个端口信息通过类似如下的方式：

```
#使用组播地址 239.0.0.5:19999
multicast://239.0.0.5:19999
```

- 每个 UDP 数据报文中，包含的主要信息包括本节点 ActiveMQ 的版本信息，以及连接到自己所需要使用的 host 名字、协议名和端口信息。类似如下：

```
default.ActiveMQ-4.aillve%localhost%auto+nio://activemq:61616
```

## （2）transportConnector 标签的关联设置

任何一个 ActiveMQ 服务节点 A 要连接到另外的 ActiveMQ 服务节点，都需要使用当前节点 A 已经公布的 transportConnector 连接端口，例如以下配置中，能够供其他服务节点进行连接的就只有两个 transportConnector 连接中的任意一个：

```
.....
<transportConnectors>
  <!-- 其他 ActiveMQ 服务节点，只能使用以下三个连接协议和端口进行连接 -->
  <!-- DOS protection, limit concurrent connections to 1000 and frame
size to 100MB -->
  <transportConnector name="tcp" uri="tcp://0.0.0.0:61614?>
```

```

maximumConnections=1000&wireFormat.maxFrameSize=104857600"/>
    <transportConnector name="nio" uri="nio://0.0.0.0:61618?
maximumConnections=1000" />
    <transportConnector name="auto" uri="auto://0.0.0.0:61617?
maximumConnections=1000" />
</transportConnectors>
.....

```

要将哪一个连接方式通过 UDP 数据报文向其他 ActiveMQ 节点进行公布，就需要在 `transportConnector` 标签上使用 `discoveryUri` 属性进行标识，如下所示：

```

.....
<transportConnectors>
    .....
    <transportConnector name="ws" uri="ws://0.0.0.0:61614?
maximumConnections=1000&wireFormat.maxFrameSize=104857600"/>
    <transportConnector name="auto" uri="auto+nio://0.0.0.0:61616?
maximumConnections=1000&wireFormat.maxFrameSize=104857600"
discoveryUri="multicast://239.0.0.5" />
</transportConnectors>
.....
<networkConnectors>
    <networkConnector uri="multicast://239.0.0.5"/>
</networkConnectors>
.....

```

### (3) 其他注意事项

- 关于防火墙：请记得关闭 Linux 服务器上对需要公布的 IP 和端口的限制；
- 关于 hosts 路由信息：由于基于组播的动态发现机制，能够找到的是目标 ActiveMQ 服务节点的机器名，而不是直接找到的 IP。所以请设置当前服务节点的 hosts 文件，以便当前 ActiveMQ 节点能够通过 hosts 文件中的 IP 路由关系，得到机器名与 IP 的映射：

```

# hosts 文件
192.168.61.139      activemq1
192.168.61.138      activemq2

```

## 4. 静态 Network Connectors

相比于基于组播发现方式的动态 Network Connectors 而言，虽然静态 Network Connectors 没有那样灵活的横向扩展性，但是却可以适用于网络环境受严格管理的情况。例如：管理员关闭了交换机/路由器的组播功能等情况。

配置静态 Network Connectors 的 ActiveMQ 集群的方式也很简单，只需要更改 `networkConnectors` 标签中的配置即可，而无须关联改动 `transportConnectors` 标签。但是配置静

态 Network Connectors 的 ActiveMQ 集群时，需要注意非常关键的细节：每一个节点都要配置其他所有节点的连接位置。

为了演示配置过程，我们假设 ActiveMQ 集群由两个节点构成，分别是 activemq1: 192.168.61.138 和 activemq2: 192.168.61.139。那么配置情况如下所示。

- 192.168.61.138: 需要配置 activemq2 的位置信息以便进行连接

```
.....
<transportConnectors>
  <transportConnector name="auto" uri="auto+nio://0.0.0.0:61616?maximum
Connections=1000&wireFormat.maxFrameSize=104857600&consumer.prefetch
Size=5"/>
</transportConnectors>
.....
<!-- 请注意，一定需要 192.168.61.139 (activemq2) 提供了这样的连接协议和端口 -->
<networkConnectors>
  <networkConnector uri="static:(auto+nio://192.168.61.139:61616)"/>
</networkConnectors>
.....
```

- 192.168.61.139: 需要配置 activemq1 的位置信息以便进行连接

```
.....
<transportConnectors>
  <transportConnector name="auto" uri="auto+nio://0.0.0.0:61616?maximum
Connections=1000&wireFormat.maxFrameSize=104857600&consumer.prefetch
Size=5"/>
</transportConnectors>
.....
<!-- 请注意，一定需要 192.168.61.138 (activemq1) 提供了这样的连接协议和端口 -->
<networkConnectors>
  <networkConnector uri="static:(auto+nio://192.168.61.138:61616)"/>
</networkConnectors>
.....
```

同理，如果你的 ActiveMQ 集群规划中有三个 ActiveMQ 服务节点，那么任何一个节点都应该配置其他两个服务节点的连接方式。在配置格式中使用“,”符号进行分割：

```
.....
<networkConnectors>
  <networkConnector
uri="static:(tcp://host1:61616,tcp://host2:61616,tcp://...)" />
</networkConnectors>
.....
```

以下是配置完成后可能的效果。

- 192.168.61.138 (activemq1) (图 9-33)

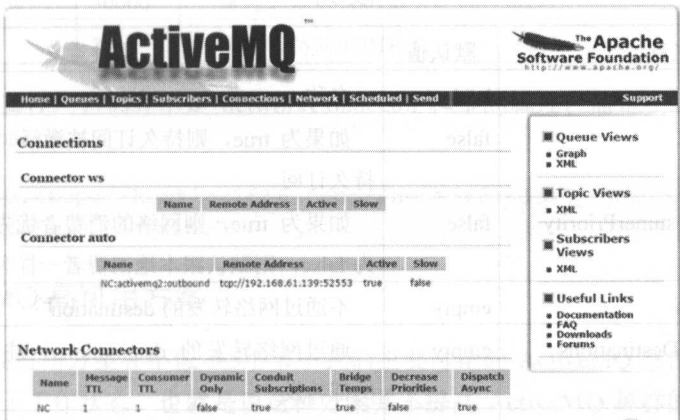


图 9-33 activemq1 效果

- 192.168.61.139 (activemq2) (图 9-34)

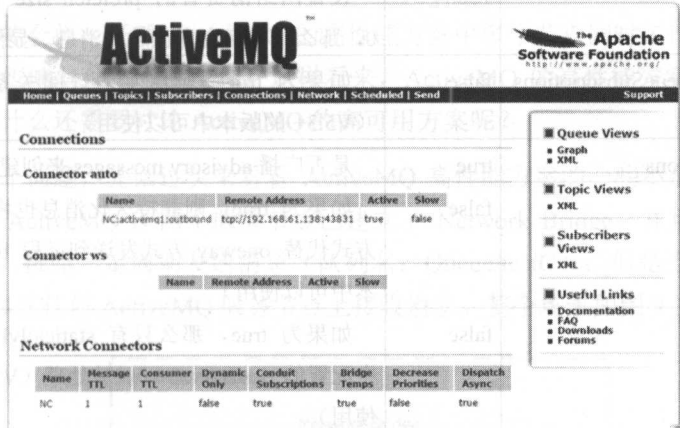


图 9-34 activemq2 效果

## 5. 其他配置属性

表 9-7 列举了在 `networkConnector` 标签中还可以使用的属性及其意义，请特别注意其中的 `duplex` 属性。如果只从字面意义理解该属性，则被称为“双工模式”；如果该属性为 `true`，当这个节点使用 **Network Bridge** 连接到其他目标节点后，那么将强制目标也建立 **Network Bridge** 进行反向连接。其目的在于让消息既能发送到目标节点，又可以通过目标节点接收消

息，但实际上大多数情况下是没有必要的，因为目标节点一般都会自行建立连接到本节点。所以该 duplex 属性的默认值为 false。

表 9-7

属性名称	默认值	属性意义
name	bridge	名称
dynamicOnly	false	如果为 true，则持久订阅被激活时才创建对应的网络持久订阅
decreaseNetworkConsumerPriority	false	如果为 true，则网络的消费者优先级降低为-5。如果为 false，则默认跟本地消费者一样为 0
excludedDestinations	empty	不通过网络转发的 destination
dynamicallyIncludedDestinations	empty	通过网络转发的 destinations，注意空列表代表所有的都转发
staticallyIncludedDestinations	empty	匹配的都将通过网络转发即使没有对应的消费者，如果为默认的“empty”，那么说明所有都要被转发
duplex	false	已经进行详细介绍的“双工”属性
prefetchSize	1000	设置网络消费者的 prefetch size 参数。如果设置成 0，那么消费者会自己轮询消息。显然这是不被建议的
suppressDuplicateQueueSubscriptions	false	如果为 true，则重复的订阅关系一产生即被阻止（V5.3+ 的版本中可以使用）
bridgeTempDestinations	true	是否广播 advisory messages 来创建临时 destination
alwaysSyncSend	false	如果为 true，则非持久化消息也将使用 request/reply 方式代替 oneway 方式发送到远程 broker（V5.6+ 的版本中可以使用）
staticBridge	false	如果为 true，那么只有 staticallyIncludedDestinations 中配置的 destination 可以被处理（V5.6+ 的版本中可以使用）

如表 9-8 所示这些属性，只能在静态 Network Connectors 模式下使用。

表 9-8

属性名称	默认值	属性意义
initialReconnectDelay	1000	重连之前等待的时间（ms）（如果 useExponentialBackOff 为 false）
useExponentialBackOff	true	如果该属性为 true，那么在每次重连失败到下次重连之前，都会增大等待时间



续表

属性名称	默认值	属性意义
maxReconnectDelay	30000	重连之前等待的最大时间（ms）
backOffMultiplier	2	增大等待时间的系数

请注意这些属性，它们并不是 `networkConnector` 标签的属性，而是在 `uri` 属性中进行设置的，例如：

```
uri="static:(tcp://host1:61616,tcp://host2:61616)?maxReconnectDelay=5000
&useExponentialBackOff=false"
```

9.5.2 ActiveMQ 高可用方案

ActiveMQ 高可用方案并不像 9.5.1 节中我们主要讨论的 ActiveMQ 高性能方案那样，同时有多个节点都处于工作状态，也就是说这种方案并不提高 ActiveMQ 集群的性能。高可用方案可以从集群中的多个节点选择一个，让其处于工作状态，集群中其他节点则处于待命状态。当工作状态的节点由于各种异常情况停止服务时，保证处于待命的节点能够无缝接替其工作。

1. ActiveMQ 高性能方案的不足

有的读者可能会问，既然 ActiveMQ 的高性能方案中多个节点同时工作，在某个节点异常的情况下也不会影响其他节点的工作。这样看来，ActiveMQ 的高性能方案已经避免了单点故障，那么我们为什么还需要讨论 ActiveMQ 的高可用方案呢？

为了回答这个问题，先回过头来看看 ActiveMQ 高性能方案的一些不足。假设如下的场景：ActiveMQ A 和 AcitveMQ B 两个服务节点已建立了 Network Bridge；并且 Producer1 连接在 ActiveMQ A 上，按照一定周期发送消息（队列名：Queue/testC）；但是当前并没有任何消费者 Consumer 连接在任何 ActiveMQ 服务节点上接收消息。整个场景如图 9-35 所示。

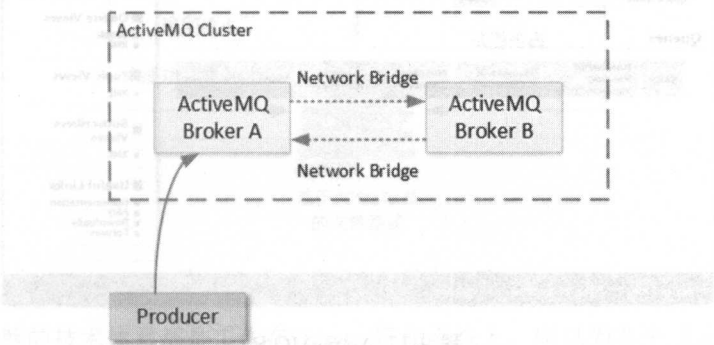


图 9-35 ActiveMQ 上的客户端连接

在发送了若干消息后，我们查看两个 ActiveMQ 服务节点的消息情况，发现 ActiveMQ A 并没有把队列 Queue/testC 中的消息同步到 ActiveMQ B。原来 AcitveMQ Network Bridge 的工作原理是：只在服务节点间传输需要传输的消息，这样做的原因是为了尽量减少 AcitveMQ 集群网络中不必要的数据流量。在我们实验的这种情况下并没有任何消费者在任何 ActiveMQ 服务节点上监听/订阅队列 Queue/testC 中的消息，所以消息并不会进行同步。

- ActiveMQ A 节点中的 Queue/testC 队列中有 10 条消息，如图 9-36 所示。

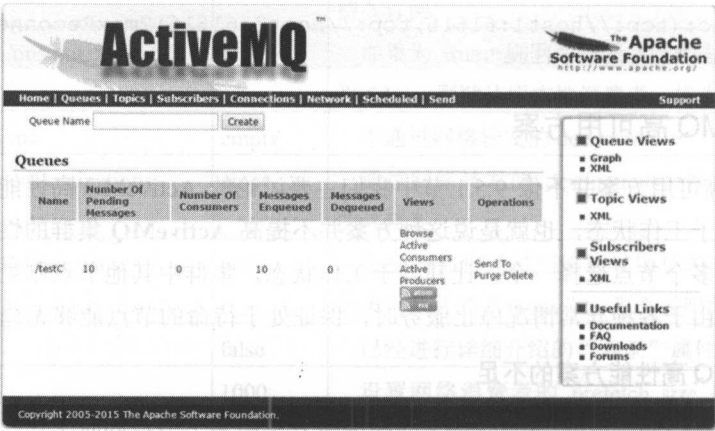


图 9-36 ActiveMQ A

- 这些消息并没有被实时传输到 ActiveMQ B 节点中，因为 B 节点中并没有任何消费者监听/订阅这个队列中的消息，如图 9-37 所示。

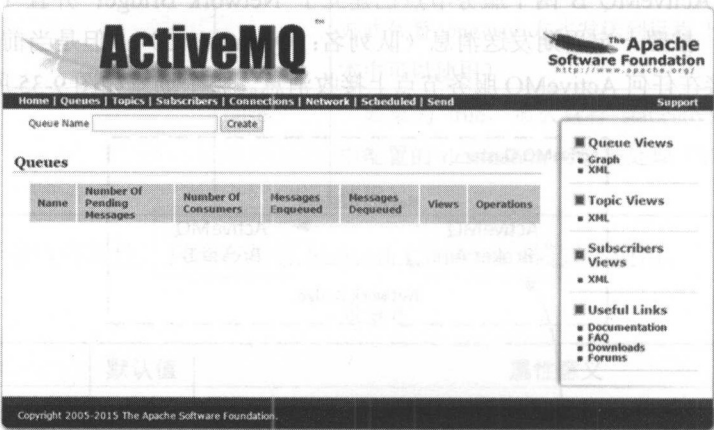


图 9-37 ActiveMQ B

那么这样的工作机制带来的问题是，当没有任何消费者在任何服务节点订阅 **ActiveMQ A** 中队列的消息时，一旦 **ActiveMQ A** 由于各种异常退出，后来的消费者就再也收不到消息，直到 **ActiveMQ A** 恢复工作。所以我们需要一种高可用方案，让某一个服务节点能够 7×24 小时地稳定提供消息服务。

2. 基于共享文件系统的高可用方案

(1) 方案介绍

基于共享文件系统的高可用方案可以说是 **ActiveMQ** 消息中间件中最早出现的一种高可用方案（图 9-38）。它的工作原理很简单：让若干个 **ActiveMQ** 服务节点，共享一个文件系统。当某一个 **ActiveMQ** 服务抢占到了这个文件系统的操作权限时，就给文件系统的操作区域加锁；其他服务节点一旦发现这个文件系统已经被加锁（并且锁不属于本进程），就会自动进入 **Salve** 模式。

**ActiveMQ** 早期的文件存储方案、**KahaDB** 存储方案、**LevelDB** 存储方案都支持这个工作模式。当某个 **ActiveMQ** 节点获取了文件系统的操作权限后，首先做的事情就是从文件系统中恢复内存索引结构：**KahaDB** 恢复 **BTree** 结构；**LevelDB** 恢复 **memTable** 结构。

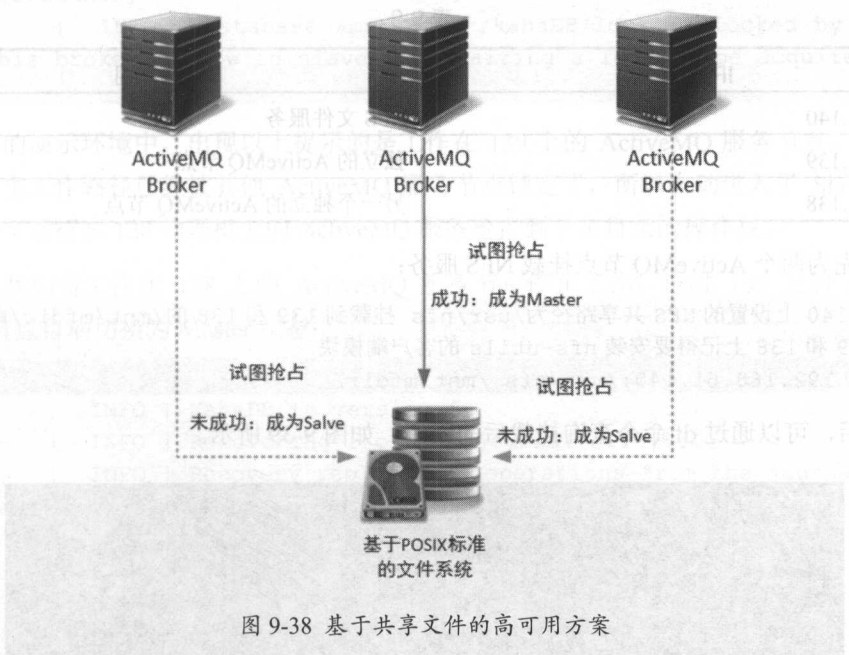


图 9-38 基于共享文件的高可用方案

因为本书讲解的技术体系都是工作在 **Linux** 操作系统上，所以为多个 **ActiveMQ** 提供共享文件系统方案的第三方文件系统都必须支持 **POSIX** 协议，这样 **Linux** 操作系统才能实现远程

挂载。

幸运的是，这样的第三方系统多不胜数，例如：基于网络文件存储的 NFS、NAS；基于对象存储的分布式文件系统 Ceph、MFS、Swift（不是 iOS 的编程语言）、GlusterFS（高版本）；以及 ActiveMQ 官方推荐的网络块存储方案：SAN。

由于本书篇幅的原因，笔者无法将博客中另一个重要的专题“系统存储”部分的知识点整理进来，关于块存储、分布式存储对象存储的知识点，在那里有详细的介绍：<http://blog.csdn.net/yinwenjie>。为了讲解简单，我们以下的讲解采用 NFS 实现文件系统的共享。NFS 技术比较成熟，在很多业务领域都有使用案例。如果业务生产环境还没有达到滴滴、大众点评等网站对文件存储性能上的要求，也可以将 NFS 用于生产环境。

(2) 示例参考

下面来演示两个 ActiveMQ 节点建立在 NFS 网络文件存储上的 Master/Slave 方案。关于怎么安装 NFS 软件就不进行介绍了，毕竟本部分内容的核心还是消息服务中间件，不清楚 NFS 安装的读者可以自行百度/Google。

表 9-9 是我们演示环境中的 IP 位置和功能。

表 9-9

IP 位置	作 用
192.168.61.140	NFS 文件服务
192.168.61.139	独立的 ActiveMQ 节点
192.168.61.138	另一个独立的 ActiveMQ 节点

- 首先为两个 ActiveMQ 节点挂载 NFS 服务：  
-- 在 140 上设置的 NFS 共享路径为 /usr/nfs 挂载到 139 和 138 的 /mnt/mfdir/ 路径下  
-- 139 和 138 上记得要安装 nfs-utils 的客户端模块  
mount 192.168.61.140:/usr/nfs /mnt/mfdir/

挂载后，可以通过 df 命令查询挂载后的结果，如图 9-39 所示。

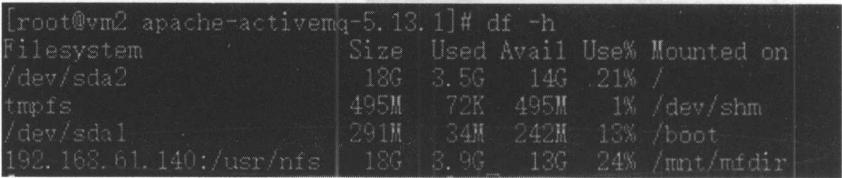


图 9-39 NFS 挂载情况

从图 9-39 中可以看到，192.168.61.140 上提供的 NFS 共享目录通过 mount 命令挂载成为 138 和 139 两个物理机上的本地磁盘路径。

- 然后更改 138 和 139 上 ActiveMQ 的主配置文件，主要目的是将使用的 KahaDB/LevelDB 的主路径设置为在共享文件系统的相同位置：

```
.....
<persistenceAdapter>
  <!--
    这里使用 KahaDB，工作路径设置在共享路径的 kahaDB 文件夹下
    138 和 139 都设置为相同的工作路径
  -->
  <kahaDB directory="/mnt/mfdir/kahaDB"/>
</persistenceAdapter>
.....
```

同时启动 138 和 139 上的 ActiveMQ 服务节点。这时可以看到某个节点出现以下的提示信息（记得是通过 console 模式进行观察）：

```
.....
jvm 1      | INFO | Using Persistence Adapter: KahaDBPersistenceAdapter
[/mnt/mfdir/kahaDB]
jvm 1      | INFO | Database /mnt/mfdir/kahaDB/lock is locked by another
server. This broker is now in slave mode waiting a lock to be acquired
.....
```

在本书的演示环境中，出现以上提示的是工作在 139 上的 ActiveMQ 服务节点。这说明这个节点发现主工作路径已经被其他 ActiveMQ 服务节点锁定了，所以自动进入了 Slave 状态。还说明另一个运行在 128 物理机上的 ActiveMQ 服务抢占到了主目录的操作权。

接下来我们将工作在 138 上的 ActiveMQ 服务节点停止工作，这时 139 上的 ActiveMQ Slave 服务节点自动切换为 Master 状态：

```
.....
jvm 1      | INFO | KahaDB is version 6
jvm 1      | INFO | Recovering from the journal @1:47632
jvm 1      | INFO | Recovery replayed 53 operations from the journal in
0.083 seconds.
jvm 1      | INFO | PListStore:[/usr/apache-activemq-5.13.1/bin/linux-
x86-64/../../data/activemq2/tmp_storage] started
jvm 1      | INFO | Apache ActiveMQ 5.13.1 (activemq2, ID:vm2-46561-
1461220298816-0:1) is starting
.....
```

从以上的提示可以看到，139 上的 ActiveMQ 节点在自己的内存区域恢复了 KahaDB 的索



引信息，并切换为 Master 状态继续工作。需要注意的是，在 139 上的 ActiveMQ 节点切换为 Master 状态后，就算之前 138 上的 ActiveMQ 节点重新恢复工作，后者也不会再获得主目录的操作权限，只能进入 Salve 状态。

### 3. 基于关系型数据库的高可用方案

基于关系型数据库的高可用方案，它的工作原理实际上和基于共享文件系统的高可用方案相似：

- 首先使用关系型数据库作为 ActiveMQ 的持久化存储方案时，在指定的数据库中会有三张数据表：activemq\_acks、activemq\_lock、activemq\_msgs（有的情况下生成的数据表名会是大写的，这是因为数据库自身设置的原因）。
- 其中“activemq\_lock”这张数据表记录了当前对数据库拥有操作权限的 ActiveMQ 服务的 ID 信息、Name 信息。各个 ActiveMQ 服务节点从这张数据表识别当前哪一个节点是 Master 状态。
- 当需要搭建高可用方案时，两个或者更多的 ActiveMQ 服务节点共享同一个数据服务。首先抢占到数据库服务的 ActiveMQ 节点，会将数据库中“activemq\_lock”数据表的 Master 状态标记为自己，这样其他 ActiveMQ 服务节点就会进入 Salve 状态。

在之前的内容中已经向读者讲述了如何进行 ActiveMQ 服务的数据库存储方案的配置，这里就不再进行赘述。只需要将每个 ActiveMQ 服务节点的数据库连接设置成相同的位置，即可完成该高可用方案的配置工作。

为了便于各位读者进行这种方案的配置实践，这里给出了关键的配置信息：实际上真得很简单，一定要首先确保数据库是可用的，并且每一个 ActiveMQ 节点都这样配置。

```
.....
<broker xmlns="http://activemq.apache.org/schema/core">
    .....
    <persistenceAdapter>
        <!-- 设置使用的数据源 -->
        <jdbcPersistenceAdapter dataSource="#mysql-ds"/>
    </persistenceAdapter>
    .....
</broker>
.....
<!-- 一定要确保数据库可用，且在 ActiveMQ 的 lib 目录中有必要的 jar 文件 -->
<bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql:// 你的 mysql 连接 URL 信
```

```
息?relaxAutoCommit=true"/>
    <property name="username" value="activemq"/>
    <property name="password" value="activemq"/>
    <property name="poolPreparedStatements" value="true"/>
</bean>
.....
```

4. LevelDB + ZooKeeper 的高可用方案

从 ActiveMQ V5.9.0+ 版本开始，ActiveMQ 为使用者提供了一种新的 Master/Salve 高可用方案。这个方案中，可以让每个节点都有自己独立的 LevelDB 数据库（并不像本节中介绍的共享文件系统那样，共享 LevelDB 的工作目录），并且使用 ZooKeeper 集群控制多个 ActiveMQ 节点的工作状态，完成 Master/Salve 状态的切换。工作模式如图 9-40 所示（摘自官网）。

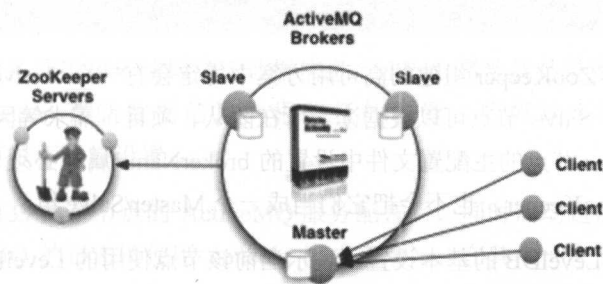


图 9-40 LevelDB + ZooKeeper（来源于 ActiveMQ 官方站点）

在这种新的工作模式下，Master 节点和各个 Salve 节点通过 ZooKeeper 进行工作状态同步，即使某个 Salve 节点新加入也没有问题。下面我们一起来讨论一下如何搭建 LevelDB + ZooKeeper 的高可用方案。表 9-10 中是我们将要使用的 IP 位置和相关位置的工作任务。

表 9-10

IP 位置	作 用
192.168.61.140	单节点状态的 ZooKeeper 服务
192.168.61.139	独立的 ActiveMQ 节点
192.168.61.138	另一个独立的 ActiveMQ 节点

- 首先更改 139 和 138 上工作的 ActiveMQ 服务节点，让它们使用独立的 LevelDB，并且都连接到 ZooKeeper 服务。

```
.....
<!--
```

注意无论是 Master 节点还是 Salve 节点，它们的 brokerName 属性都必须一致  
 否则 ActiveMQ 集群就算连接到了 ZooKeeper，也不会把它们当成一个 Master/Salve 组  
 -->

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="activemq"
dataDirectory="${activemq.data}">
    .....
    <persistenceAdapter>
        <replicatedLevelDB
            directory="/usr/apache-activemq-5.13.1/data/levelDB"
            replicas="1"
            bind="tcp://0.0.0.0:61615"
            zkAddress="192.168.61.140:2181"
            zkPath="/activemq/leveldb"/>
        </persistenceAdapter>
    .....
</broker>
.....
```

通过 LevelDB + ZooKeeper 组建的高可用方案中肯定会有一个 ActiveMQ 节点充当 Master 节点，至于有多少个 Salve 节点可以根据读者所在团队、项目、需求等因素来综合考虑了。这些 Master 节点和 Salve 节点的主配置文件中设置的 brokerName 属性必须一致，否则 ActiveMQ 集群就算连接到了 ZooKeeper，也不会把它们当成一个 Master/Salve 组。

directory 属性是 LevelDB 的基本设置，表示当前该节点使用的 LevelDB 所在的工作目录，由于每个节点都有其独立运行的 LevelDB，所以各个节点的 directory 属性设置的目录路径可以不一样。但是根据对正式环境的管理经验，建议还是将每个节点的 directory 属性设置成相同的目录路径。对于 replicas 属性，官方给出的解释如下：

The number of nodes that will exist in the cluster. At least  $(replicas/2)+1$  nodes must be online to avoid service outage. (default: 3)

这里的“number of nodes”包括了 Master 节点和 Salve 节点的总和。换句话说，如果集群中一共有 3 个 ActiveMQ 节点，且只允许最多有一个节点出现故障。那么这里的值可以设置为 2（当然设置为 3 也行，因为整型计算中  $3/2+1=2$ ）。但如果将 replicas 属性设置为 4，就代表不允许 3 个节点的任何一个节点出错，因为： $(4/2)+1=3$ ，也就是说 3 个节点是本集群正常工作时能够允许的最小节点数。

一旦 ZooKeeper 发现当前集群中可工作的 ActiveMQ 节点数小于所谓的“ $At\ least\ (replicas/2)+1\ nodes$ ”，在 ActiveMQ Master 节点的日志中就会给出提示：“Not enough cluster members connected to elect a master”，然后整个集群都会停止工作，直到有新的节点连入，并

达到所规定的“**At least (replicas/2)+1 nodes**”数量。

`bind` 属性指明了当本节点成为一个 Master 节点后，通过哪一个通信位置进行和其他 Slave 节点的消息复制操作。注意这里配置的监听地址和端口不能在 `transportConnectors` 标签中进行重复配置，否则节点在启动时会报错。

When this node becomes a master, it will bind the configured address and port to service the replication protocol. Using dynamic ports is also supported.

`zkAddress` 属性指明了连接的 ZooKeeper 服务节点所在的位置。在以上实例中由于我们只有一个 ZooKeeper 服务节点，所以只配置了一个位置。如果有多个 ZooKeeper 服务节点，那么请依次配置这些 ZooKeeper 服务节点的位置，并以“,”进行分隔：

```
zkAddress="zoo1.example.org:2181,zoo2.example.org:2181,zoo3.example.org:2181"
```

由于 ZooKeeper 服务使用树形结构描述数据信息，`zkPath` 属性就是设置整个 ActiveMQ 主/备方案集群在 ZooKeeper 存储数据信息的根路径的位置。当然这个属性有一个默认值“/default”，所以也可以不进行设置。

- 在完成 138 和 139 两个节点的 ActiveMQ 服务配置后，同时启动这两个节点（注意，为了观察 ActiveMQ 的日志请使用 `console` 模式启动）。在其中一个节点上，可能会出现以下日志信息：

```
.....
jvm 1    | INFO | Opening socket connection to server 192.168.61.140/
92.168.61.140:2181
jvm 1    | INFO | Socket connection established to 192.168.61.140/
92.168.61.140:2181, initiating session
jvm 1    | INFO | Session establishment complete on server 192.168.61.
40/192.168.61.140:2181, sessionId = 0x1543b74a86e0002, negotiated timeout =
4000
jvm 1    | INFO | Not enough cluster members have reported their update
positions yet.
jvm 1    | INFO | Promoted to master
jvm 1    | INFO | Using the pure java LevelDB implementation.
jvm 1    | INFO | Apache ActiveMQ 5.13.1 (activemq, ID:vm2-45190-
1461288559820-0:1) is starting
jvm 1    | INFO | Master started: tcp://activemq2:61615
.....
```

从以上的日志可以看到，这个节点连接上了 ZooKeeper，并且分析了当前 ZooKeeper 上已



连接的其他节点状态后（实际上这个时候，还没有其他节点进行连接），将自己“提升为 Master”状态。在另外一个 ActiveMQ 的节点日志中，读者可以发现另一种形式的日志提示，类似如下：

```
.....
jvm 1 | INFO | Opening socket connection to server 192.168.61.140/
92.168.61.140:2181
jvm 1 | INFO | Socket connection established to 192.168.61.140/
92.168.61.140:2181, initiating session
jvm 1 | INFO | Session establishment complete on server 192.168.61.
40/192.168.61.140:2181, sessionId = 0x1543b74a86e0005, negotiated timeout =
4000
jvm 1 | INFO | Slave started
.....
```

从日志中可以看到，这个节点成为一个 Slave 状态的节点。

接下来尝试停止当前 Master 节点的工作，并且观察当前 Slave 节点的状态变化。注意，**replicas** 属性的值一定要进行正确的设置：如果当 Master 节点停止后，当前还处于活动状态的节点总和小于 “(replicas/2) + 1”，那么整个集群都会停止工作！

```
.....
jvm 1 | INFO | Not enough cluster members have reported their update
positio ns yet.
jvm 1 | INFO | Slave stopped
jvm 1 | INFO | Not enough cluster members have reported their update
positio ns yet.
jvm 1 | INFO | Promoted to master
jvm 1 | INFO | Using the pure java LevelDB implementation.
jvm 1 | Replaying recovery log: 5.364780% done (956,822/17,835,250
bytes) @ 1 03,128.23 kb/s, 0 secs remaining.
jvm 1 | Replaying recovery log: 9.159451% done (1,633,611/17,835,250
bytes) @ 655.68 kb/s, 24 secs remaining.
jvm 1 | Replaying recovery log: 23.544615% done (4,199,241/17,835,250
bytes) @ 2,257.21 kb/s, 6 secs remaining.
jvm 1 | Replaying recovery log: 89.545681% done
(15,970,696/17,835,250 bytes) @ 11,484.08 kb/s, 0 secs remaining.
jvm 1 | Replaying recovery log: 100% done
jvm 1 | INFO | Master started: tcp://activemq1:61615
.....
```

从以上的日志片段可以看到，Slave 节点接替了之前 Master 节点的工作状态，并恢复之前已同步的 LevelDB 文件到本节点的本地内存中，继续进行工作。

我们在本节做的演示只有两个 ActiveMQ 服务节点和一个 ZooKeeper 服务节点，主要是为



了向读者介绍 ActiveMQ 下 LevelDB + ZooKeeper 的高可用方案的配置和切换过程，说明 ActiveMQ 高可用方案的重要性。在实际生产环境中，这样的节点数量配置会显得很单薄，特别是在 ZooKeeper 服务节点只有一个的情况下是不能保证整个集群稳定工作的。正式环境下，建议至少使用三个 ZooKeeper 服务节点和三个 ActiveMQ 服务节点，并将 `replicas` 属性设置为 2。

### 5. ActiveMQ 客户端的故障转移

以上三种高可用方案，都已向各位读者介绍完了。细心的读者会发现一个问题，因为我们没有使用类似 Keepalived 那样的第三方软件支持浮动 IP。那么无论使用以上三种高可用方案的哪一种，虽然服务端可以无缝切换提供连续的服务，但是对于客户端来说连接服务器的 IP 都会发生变化。也就是说客户端都会因为连接异常脱离正常工作状态。

为了解决这个问题，ActiveMQ 的客户端连接提供了配套的解决办法：连接故障转移。开发人员可以预先设置多个可能进行连接的 IP 位置（这些位置不一定同时都是可用的），ActiveMQ 的客户端会从这些连接位置选择其中一个进行连接，当连接失败时自动切换到另一个位置连接。使用方式类似如下：

```
//这样的设置，即使在发送/接收消息的过程中出现问题，客户端连接也会进行自动切换
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory
("failover:(tcp://192.168.61.138:61616,tcp://192.168.61.139:61616)");
```

## 9.6 其他 MQ 技术：Apache Kafka

Apache Kafka 最初由 LinkedIn 贡献，目前它是 Apache 下的一个顶级开源项目。**Apache Kafka** 设计的首要目标是解决 LinkedIn 网站中海量的用户操作行为记录、页面浏览记录，后继的 Apache Kafka 版本也都是将“满足高数据吞吐量”作为版本优化的首要目标。为了达到这个目标，Apache Kafka 甚至在其他功能方面上做了一定的牺牲，例如消息的事务性。如果系统需要进行单位时间内大量的数据采集工作，而可以容忍一定的消息错误率，那么可以考虑在系统设计方案中加入 Apache Kafka。

### 9.6.1 Kafka 设计概要

一个完整的 Apache Kafka 解决方案的组成包括四个要素：Producer（消息生产者）、Server Broker（服务代理器）、ZooKeeper（协调者）、Consumer（消息消费者）。**Apache Kafka** 在设计之初就被认为是集群化工作的，所以说清楚 Apache Kafka 的设计结构除了要讲述每一个 Kafka Broker 是如何工作的，还要讲述清楚整个 Apache Kafka 集群是如何工作的。

### 1. Kafka Broker 工作结构 (图 9-41)

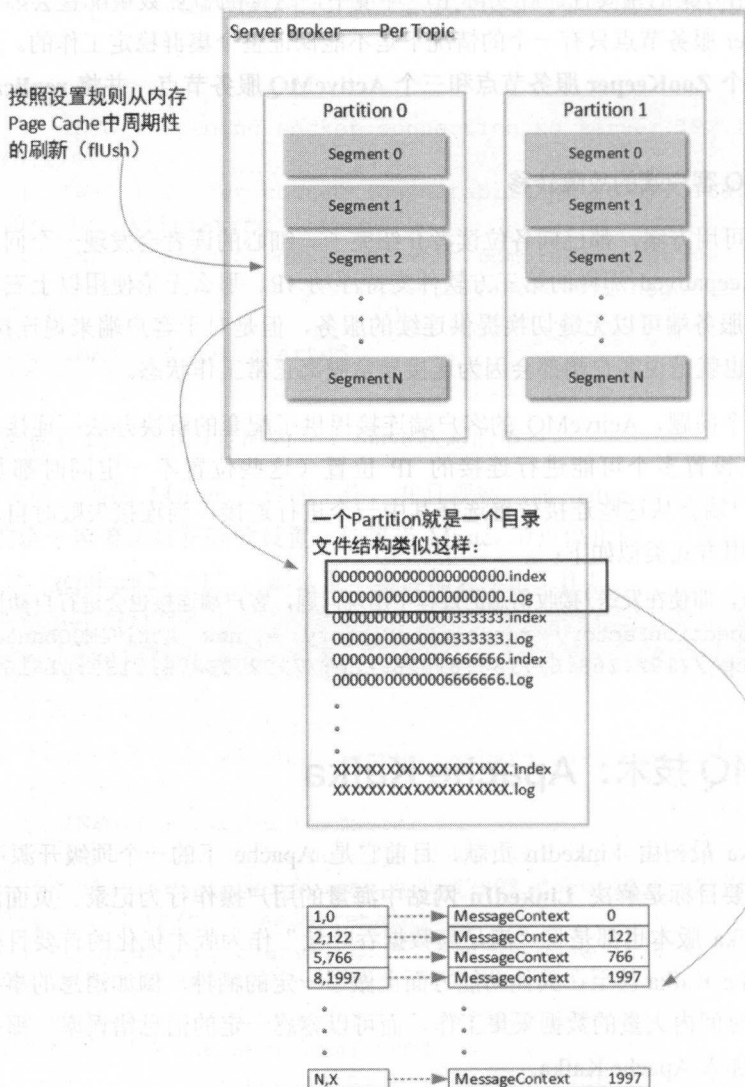


图 9-41 Kafka Broker 工作结构

在 Apache Kafka 的 Server Broker 设计中，一个独立进行消息获取、消息记录和消息发送的操作队列称之为 Topic（和 ActiveMQ 中 Queue 或者 Topic 的概念同属一个级别）。下面我们讨论的内容都是针对一个 Topic 而言，后续内容就不再进行重复强调了。

- 图 9-41 描述了一个独立的 Topic 构造结构：Apache Kafka 将 Topic 拆分成多个分区

(Partition)，这些分区 (Partition) 可能存在于同一个 Broker 上也可能存在于不同的 Broker 上。如果仔细观察 Kafka 的文件存储结构就会发现 Kafka 会为 Topic 中每一个分区创建一个独立的文件夹，类似如下所示，以下的 Topic—my\_topic2 一共创建了 4 个分区：

```
[root@kafka1 my_topic2-0]# ls
drwxr-xr-x. 2 root root 4096 4月 29 18:32 my_topic2-0
drwxr-xr-x. 2 root root 4096 4月 29 18:32 my_topic2-1
drwxr-xr-x. 2 root root 4096 4月 29 18:32 my_topic2-2
drwxr-xr-x. 2 root root 4096 4月 29 18:32 my_topic2-3
```

- 由 Producer 发送的消息会被分配到各个分区 (Partition) 中进行存储，至于它们是按照什么样的规则被分配的在后面会进行讲述。一条消息记录只会被分配到一个分区进行存储，并且这些消息以分区为单位保持顺序排列。这些分区是 Apache Kafka 性能的第一种保证方式：单位数量相同的消息将分发到存在于多个 Broker 服务节点上的多个 Partition 中，并利用每个 Broker 服务节点的计算资源进行独立处理。
- 每一个分区都中会有一个或者多个段 (segment) 结构。如图 9-41 所示，一个段 (segment) 结构包含两种类型的文件：.index 后缀的索引文件和.log 后缀的数据文件。前一个 index 文件记录了消息在整个 Topic 中的序号以及消息在 Log 文件中的偏移位置 (offset)，通过这两个信息，Kafka 可以在后一个 Log 文件中找到这条消息的真实内容。
- 在磁盘上进行的文件操作只有采用顺序读和顺序写才能做到高效的磁盘 I/O 性能。这是 Kafka 保证性能的又一种方式。对索引文件始终保证顺序读写：当在磁盘上记录一条消息时，始终在文件的末尾进行操作；当在磁盘上读取一条消息时，通过 Index 顺序查找到消息的 offset 位置，再进行消息读取。后一种消息读取操作下，如果 Index 文件过大，Kafka 的磁盘操作就会耗费掉相当的时间。所以 Kafka 需要对 Index 文件和 Log 文件进行分段。
- 实际上，Kafka 之所以“快”，并不只是因为它的 I/O 操作是顺序读写并拥有多个分区的概念；毕竟类似于 ActiveMQ 也有多节点集群的概念，并且后者通过使用 LevelDB 或者 KahaDB 这样的存储方案也可以实现磁盘的顺序 I/O 操作。要知道如果消息消费者真正需要到磁盘上寻找数据了，那么整个 Kafka 集群的性能也不会好到哪儿去：目前 SATA3 串口通信的理论速度也只有 6Gbps，使用 SATA3 串口通信的固态硬盘，真实的顺序读取最快速度也不过 550Mbps/s。
- Kafka 对 Linux 操作系统下 Page Cache 技术的应用，才是其高性能的最大保证。真实的业务环境下 Kafka 一般不需要在磁盘上为消费者寻找消息记录，前提是要保证内存空间够大。关于 Linux 操作系统下的 Page Cache 技术又是另外一个技术话题，笔者会在自己博客的“系统存储”专题进行讲解，这里不再扩展讲解了。

## 2. Kafka Cluster 结构

讲解了单个 Kafka Broker 结构后，再看看整个 Kafka 集群是怎样工作的。如图 9-42 所示视图描述了某个 Topic 下的一条消息是如何在 Kafka 集群结构中流动的（图 9-42 中的实线有向箭头）。

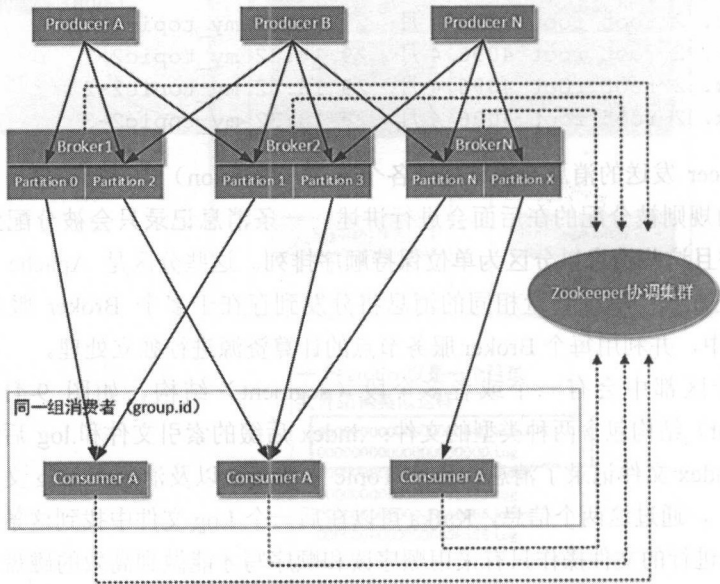


图 9-42 Kafka Cluster 结构

- 整个 Kafka 集群中，可以有多个消息生产者。这些消息生产者可能在同一个物理节点上，也可能在不同的物理节点。它们都必须知道哪些 Kafka Broker List 是信息可以发送的目标：消息生产者会自行决定发送的消息将会送入 Topic 的哪一个分区（Partition）。
- 消费者都是按照“组”的单位进行消息隔离：在同一个 Topic 下，Apache Kafka 会为不同的消费者组创建独立的 index 索引定位。也就是说当消息生产者发送一条消息后，同一个 Topic 下不同组的消费者都会收到这条信息。
- 同一组下的消息消费者可以消费 Topic 下一个分区或者多个分区中的消息，但是一个分区中的消息同一时间只能被同一组下的某一个消息消费者所处理。也就是说，如果某个 Topic 下只有一个分区，就不能实现消息的负载均衡。另外 Topic 下的分区数量也只能是固定的，不可以在使用过程中动态改变，这些分区在 Topic 被创建时使用命令行指定或者参考 Broker Server 中配置的默认值。
- 由于存在以上的操作规则，所以 Kafka 集群中 Consumer（消费者）需要和 Kafka 集群中的 Server Broker 进行协调工作：这个协调工作交给了 ZooKeeper 集群。ZooKeeper 集

群需要记录/协调的工作包括：当前整个 Kafka 集群中有哪些 Broker 节点以及每一个节点处于什么状态（活动/离线/状态）、当前集群中所有已创建的 Topic 以及分区情况、当前集群中所有活动的消费者组/消费者、每一个消费者组针对每个 Topic 的索引位置等。

- 当一个消费者上线，并且在消费消息之前。首先会通过 ZooKeeper 协调集群获取当前消费组中其他消费者的连接状态，并得到当前 Topic 下可用于消费的分区和该消费者组中其他消费者的对应关系。如果当前消费者发现 Topic 下所有的分区都已经有了对应的消费者了，就将自己置于挂起状态（和 Broker、ZooKeeper 的连接还是会建立，但是不会到分区 Pull 消息），以便在其他消费者失效后进行接替。
- 如果当前消费者连接时，发现整个 Kafka 集群中存在一个消费者（记为消费者 A）关联 Topic 下多个分区的情况，且消费者 A 处于繁忙无法处理这些分区下新的消息（即消费者 A 的上一批 Pull 的消息还没有处理完成）。这时新的消费者将接替原消费者 A 所关联的一个（或者多个）分区，并且一直保持和这个分区的关联。
- 由于 Kafka 集群中只保证同一个分区（Partition）下消息队列中消息的顺序。所以当一个或者多个消费者分别 Pull 多个消息分区时，你在消费者端观察的现象可能就是消息顺序是混乱的。这里我们一直在说消费者端的 Pull 行为，是指的 Topic 下分区中的消息并不是由 Broker 主动推送到（Push）到消费者端，而是由消费者端主动拉取（Pull）。

### 3. Kafka 消息复制功能

我们已经讨论了 Kafka 使用分区的概念存储消息，一个 Topic 可以有多个分区，它们分布在整个 Kafka 集群的多个 Broker 服务节点中，并且一条消息只会按照消息生产者的要求进入 Topic 的某一个分区。那么问题来了：如果某个分区中的消息在被消费端 Pull 之前，承载该分区的 Broker 服务节点就因为各种异常原因崩溃了，那么在这个 Broker 重新启动前，消费者就无法收到消息了。

为了解决这个问题，Apache Kafka 在 V 0.8+ 版本中加入了复制功能：让 Topic 下的每一个分区存储到多个 Broker 服务节点上，并由 ZooKeeper 统一管理它们的状态（图 9-43）。

请注意 Kafka 中 Partition（分区）和 replication（复制）是两个完全不同的功能，虽然它们都和“如何存储消息”这件事情有关：前者是说将若干条消息按照一定的规则分别存放在不同的区域，一条消息只存入一个区域（且多个分区可以存在于同一个 Broker 上）；后者是说，为了保证消息在被消费前不会丢失，需要将某一个区域中的消息集合复制出多个副本，同一个分区的多个副本不能存放在同一个 Broker 上。



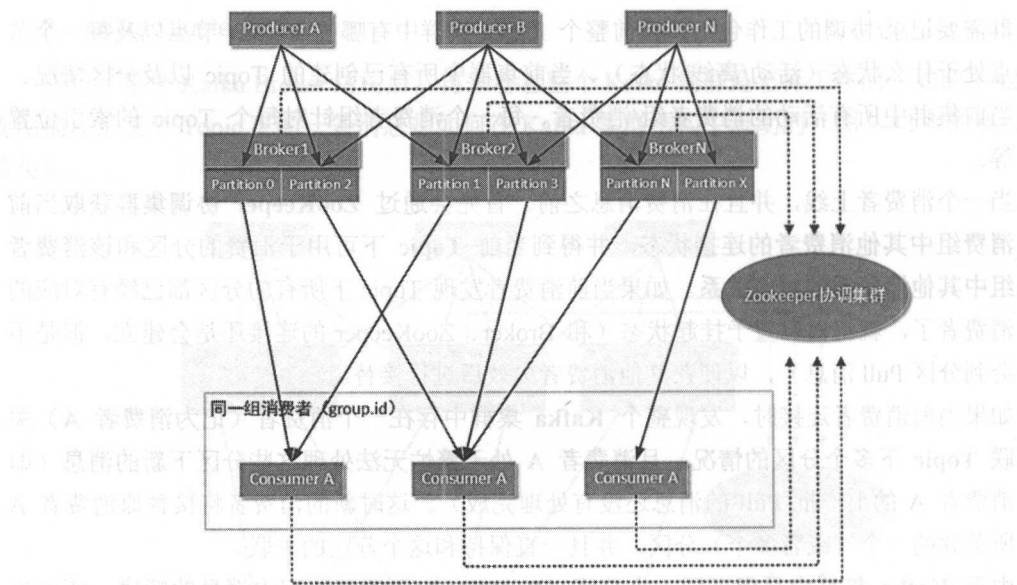


图 9-43 Kafka 中的复制

Kafka 将分区多个副本分为两种角色：Leader 和 Follower，Leader Broker 是主要服务节点，消息只会从消息生产者发送给 Leader Broker，消息消费者也只会从 Leader Broker 中 Pull 消息。Follower Broker 为从服务节点，正常情况下不会公布给生产者或者消费者直接进行操作。Follower Broker 服务节点将会主动从 Leader Broker 上 Pull 消息。

在这种工作机制下，Follower 和 Leader 的消息复制过程由于 Follower 服务节点的性能、压力、网络等原因，它们和 Leader 服务节点会有一个消息差异性。当这个差异性扩大到一定的范围时，Leader 节点就会认为这个 Follower 节点再也跟不上自己的节奏，导致的结果就是 Leader 节点会将这个 Follower 节点移出“待同步副本集”ISR (In-Sync Replicas)，不再关注这个 Follower 节点的同步问题。

只有当 ISR 中所有分区副本全部完成了某一条消息的同步过程时，这条消息才算真正完成了“记录”操作。只有这样的消息才会发送给消息消费者。至于这个真正完成“记录”操作的通知是否返回给消息生产者，完全取决于消息生产者采用的 acks 模式。

现在我们可以回过头看看上文给出的“查看 Topic 状态”命令以及命令结果：

```
# 脚本命令范例
kafka-topics.sh --describe --zookeeper 192.168.61.139:2181 --topic my_topic2
# 显示的结果
Topic:my_topic2 PartitionCount:4 ReplicationFactor:2 Configs:
```

```

Topic: my_topic2 Partition: 0 Leader: 2 Replicas: 2,1 Isr: 2,1
Topic: my_topic2 Partition: 1 Leader: 1 Replicas: 1,2 Isr: 1,2
Topic: my_topic2 Partition: 2 Leader: 2 Replicas: 2,1 Isr: 2,1
Topic: my_topic2 Partition: 3 Leader: 1 Replicas: 1,2 Isr: 1,2

```

以上命令用于显示 Topic 的基本状态信息。Partition 表示分区号，Replicas 表示所有副本的所在位置的 Broker.id 信息，Isr 表示当前状态正常可以进行消息复制的副本所在位置的 Broker.id 信息。

那么从命令结果来看，名叫“my\_topic2”的 Topic 一共有 4 个数据分区，每一个分区有两个副本。其中：0 号分区的 Leader Broker 服务节点的 id 为 2，0 号分区的两个副本分别在 id 为 2 和 id 为 1 的 Broker 服务节点上，且 id 为 2 和 id 为 1 的 Broker 上的副本状态都是正常的；同理，1 号分区的 Leader Broker 服务节点的 id 为 1，1 号分区的两个副本分别在 id 为 2 和 id 为 1 的 Broker 服务节点上，且 id 为 2 和 id 为 1 的 Broker 上的副本状态都是正常的。

#### 4. Kafka 生产者和消费者

请注意之前我们给出的 Kafka 集群方案的示意图，在图中消息生产者并没有连接到 ZooKeeper 协调服务，而是直接和多个 Kafka Server Brokers 建立了连接。和其他种类的消息队列的设计不同，在 Apache Kafka 中消息生产者（Producer）会有很多重要规则的决定权，例如：

- 消费生产者（Producer）可以决定向指定的 Topic 的哪一个分区（Partition）发送消息。而不是由 Broker 来决定。
- 消息生产者（Producer）可以决定消息达到 Kafka Broker 后，Producer 对消息的一致性关注到什么样的级别，又或者根本不关心消息在 Broker 上的一致性问题。
- 消息生产者（Producer）可以决定是以同步方式（sync）还是异步方式（async）向 Broker Server List 发送消息。在异步方式下，消费生产者（Producer）还可以决定以什么样的间隔（周期）向 Broker Server List 发送消息。
- 消息生产者（Producer）还可以随机选定 Broker Server List 中某一个服务节点，读取当前 Topic 下的分区和复制表信息，并保存在本地 Pool 中。
- 另外，Apache Kafka 中的消息生产者没有类似 ActiveMQ 中那样的事务机制。这样的设计和 Kafka 主要的业务场景有关——用来收集各种操作日志。这样的场景对消息的可靠性要求并不高：漏掉一两条日志并不影响后端大数据平台对日志数据的分析结果；而且这样的设计大量简化了 Broker 的设计结构：它不需要像 ActiveMQ 那样为达成传输但还未进行 commit 的消息专门创建存储区域“transaction store”，并在进行了 commit 或者 rollback 操作后进行标记。这种处理机制是 Apache Kafka 高效性能的又一种保障。
- Kafka 中的多个消息生产者（Producer）并不需要 ZooKeeper 服务中的任何信息为它们协调发送过程，因为没有什么可协调的。生产者唯一需要知道的是 Topic 有多少个分区

以及每个分区分别存在于哪些, Broker 上的信息都是来源于对某一个 Broker 的直接查询。所以 Kafka 集群中只剩下了 Broker 和 Consumer 需要进行协调。

这是分布式系统建设思想中一个重要的原则——不可滥用协调装置:完成同一件工作时,协调  $N$  个参与角色要比协调  $N-1$  个参与角色耗费更多的时间和性能;所以,只协调需要协调的角色,只通知需要通知的事件,只为协调过程存储必要的数据库。

### (1) 基本使用

下面的代码使用 Kafka 的 Java Client API 演示消息生产者的使用。这里我们使用的 Kafka Java Client API 的版本是 V0.8.2.2:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.10</artifactId>
  <version>0.8.2.2</version>
</dependency>
```

以下是 Kafka 消息生产者的代码,之前我们已经通过 Kafka 的命令脚本创建了一个拥有 4 个分区的 Topic—my\_topic2:

```
.....
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
//Kafka 消息生产者演示,
public class KafkaProducer {
    public static void main(String[] args) throws RuntimeException {
        Properties props = new Properties();
        //指定 Kafka 节点列表,不需要由 ZooKeeper 进行协调
        //并且连接的目的也不是为了发送消息
        //而是为了在这些节点列表中选取一个,来获取 Topic 的分区状况
        props.put("metadata.broker.list", "192.168.61.138:9092");
        //使用这个属性可以指定“将消息送到 Topic 的哪一个分区(partition)中”
        //如果业务规则比较复杂的话则可以指定分区控制器
        //不过开发者最好要清楚 Topic 有多少个分区,这样才好进行多线程(负载均衡)发送
        //props.put("partitioner.class", "kafkaTQ.PartitionerController");
        //可以通过这个参数控制是异步发送还是同步发送(默认为“同步”)
        //props.put("producer.type", "async");
        //可以通过这个属性控制复制过程的一致性规则
        //props.put("request.required.acks", "1");
        ProducerConfig config = new ProducerConfig(props);
        //创建消费者
        Producer<byte[], byte[]> producer = new Producer<byte[], byte[]>
(config);
```

```

// 由于我们为 Topic 创建了 4 个 partition, 所以将数据分别发往这 4 个分区
for (Integer partitionIndex = 0; ; partitionIndex++) {
    Date time = new Date();
    // 创建和发送消息, 可以指定这条消息的 key
    // Producer 根据这个 key 来决定这条消息发送到哪个 Partition 中
    // 另外一个可以决定 Partition 的方式是实现 kafka.producer.
    // Partitioner 接口
    String messageContext_Value = "this message from producer 由
producer 指的 partitionIndex: [" + partitionIndex % 4 + "]" + time.getTime();
    System.out.println(messageContext_Value);
    byte[] messageContext = messageContext_Value.getBytes();
    byte[] key = partitionIndex.toString().getBytes();
    // 这是消息对象
    KeyedMessage<byte[],byte[]>message = new KeyedMessage<byte[],
byte[]>("my_topic2", key , partitionIndex % 4 , messageContext);
    producer.send(message);
    // 休息 0.5 秒钟, 循环发消息
    synchronized (KafkaProducer.class) {
        try {
            KafkaProducer.class.wait(500);
        } catch (InterruptedException e) {
            e.printStackTrace(System.out);
        }
    }
}
}
}
}

```

## (2) 生产者指定分区

开发人员可以在消息生产者端指定发送的消息将要传送到 Topic 下的哪一个分区 (Partition), 但前提条件是开发人员清楚这个 Topic 有多少个分区, 否则开发人员就不知道怎么编写代码了。当然开发人员也可以完全忽略决定分区的规则, 这时将由消费者端携带的一个默认规则决定。

开发人员可以有两种方式指定分区: 第一种方法是以上代码片段中演示的那样, 在创建消息对象 KeyedMessage 时, 指定方法中 partKey/key 的值; 另一种方式是重新实现 kafka.producer.Partitioner 接口, 以便覆盖掉默认实现。

使用 KeyedMessage 类构造消息对象时, 可以指定 4 个参数, 它们分别是: Topic 名称、消息 Key、分区 Key 和 Message 消息内容。Topic 名称和 Message 消息内容很容易理解, 但是怎样理解消息 Key 和分区 Key 呢? 以下是 KeyedMessage 类的源代码 (Scala 语言):

```
package kafka.producer
```



```

.....
//A topic, key, and value.
//If a partition key is provided it will override the key for the
//purpose of partitioning but will not be stored.
case class KeyedMessage[K, V](val topic: String, val key: K, val partKey:
Any, val message: V) {
  if(topic == null)
    throw new IllegalArgumentException("Topic cannot be null.")
  def this(topic: String, message: V) = this(topic, null.asInstanceOf[K],
null, message)
  def this(topic: String, key: K, message: V) = this(topic, key, key,
message)
  def partitionKey = {
    if(partKey != null)
      partKey
    else if(hasKey)
      key
    else
      null
  }
  def hasKey = key != null
}

```

KeyedMessage 类的构造函数中有一个局部变量：partitionKey，在 KeyedMessage 类的首行注释中，对该变量进行了一个说明：

If a partition key is provided it will override the key for the purpose of partitioning but will not be stored.

从源码中可以看出，partitionKey 优先使用 partKey 作为分区依据，如果 partKey 没有被赋值，则使用 Key 作为分区依据。所以在使用 KeyedMessage 类的构造函数时，partKey 和 Key 只需要指定其中的一个就完全够了。

还可以实现 kafka.producer.Partitioner 接口，并在创建消费者对象时进行指定，以便实现分区的指定（如果不进行指定，默认的实现类为“kafka.producer.DefaultPartitioner”）。代码片段如下：

```

package kafkaTQ;
import kafka.producer.Partitioner;
import kafka.utils.VerifiableProperties;
public class PartitionerController implements Partitioner {
  //必须要有这个构造函数
  public PartitionerController(VerifiableProperties vp) {
  }
  public int partition(Object parKey, int partition) {

```



```
//在这里可以根据自身的业务过程重新运算一个分区，并进行返回。  
Integer parKeyValue = (Integer)parKey;  
return parKeyValue;  
}  
}
```

需要实现的分区方法中，第一个参数是在创建消息时所传递的 `partyKey`，第二个参数是 `send` 方法根据自身内部机制决定的目标分区。

5. Kafka 同步和异步发送

消息生产者还可以决定是以同步方式向 `Broker` 发送消息还是以异步方式向 `Broker` 发送消息。只需要使用生产者配置中的“`producer.type`”属性进行指定。当该属性值为“`sync`”时，表示使用同步发送的方式；当该属性值为“`async`”时，表示使用异步发送方式。

在异步发送方式下，开发人员调用 `send` 方法发送消息时，这个消息并不会立即被发送到 `Topic` 指定的 `Leader Partition` 所在的 `Broker`，而是会存储在本地的一个缓冲区域（一定注意是客户端本地）。当缓冲区状态满足最长等待时间或者最大数据量条数时，消息会以一个设置值批量发送给 `Broker`。如图 9-44 所示。

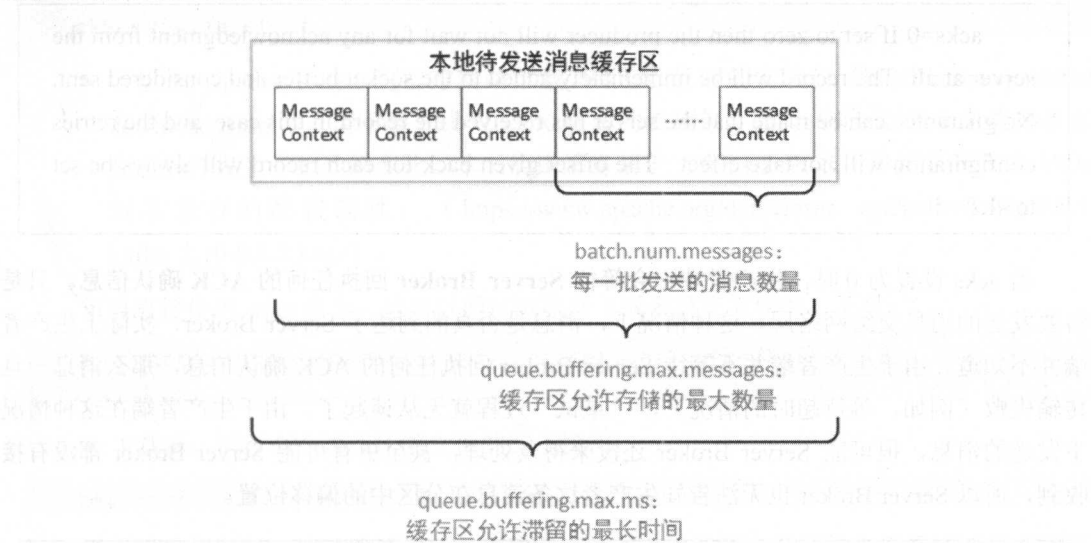


图 9-44 异步参数与定义范围

缓存区的数据按照 `batch.num.messages` 设置的数值被一批一批地发送给目标 `Broker`（默认为 200 条），如果消息的滞留时间超过了 `queue.buffering.max.ms` 设置的值（单位毫秒，默认值为 5000）就算没有达到 `batch.num.messages` 的数值，消息也会被发送。

如果由于 Broker 的原因导致消息发送缓慢,这时在本地待发送消息缓存区中的消息就有可能达到 `queue.buffering.max.messages` 设置的缓存区允许存储的最大消息数量,一旦达到这个数量,消息生产者端再次调用 `send` 方法的时候, `send` 方法所在线程就会被阻塞,直到缓存区有足够的空间能够放下新的数据为止。

## 6. Kafka 强一致性复制和弱一致性复制

Kafka 中的消息生产者还可以配置发送的消息在 Broker 端以哪种方式进行副本复制:强一致性复制还是弱一致性复制,又或者不关注消息的一致性。

在 Kafka 的实现中,强一致性复制是指当 Leader Partition 收到消息后,将在所有 Follower partition 完成这条消息的复制后才认为消息处理成功,并向消息生产者返回 ACK 信息;弱一致性复制是指当 Leader partition 收到消息后,只要 Leader Broker 自己完成了消息的存储就认为消息处理成立,并向消息生产者返回 ACK 信息(复制过程随后由 Broker 节点自行完成)。

你可以通过消息生产者配置中的“`request.required.acks`”属性来设置消息的复制性要求。在官方文档中,对于这个属性的解释如下。

`acks=0` If set to zero then the producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the retries configuration will not take effect. The offset given back for each record will always be set to -1.

当 `acks` 设置为 0 时,生产者端不会等待 Server Broker 回执任何的 ACK 确认信息。只是将要发送的消息交给网络层。这种情况下,消息是否真的到达了 Server Broker,实际上生产者端并不知道。由于生产者端并不等待 Server Broker 回执任何的 ACK 确认信息,那么消息一旦传输失败(例如,等待超时的情况),“重试”过程就无从谈起了。由于生产者端在这种情况下发送的消息,很可能 Server Broker 还没来得及处理,甚至更有可能 Server Broker 都没有接收到,所以 Server Broker 也无法告知生产者这条消息在分区中的偏移位置。

`acks=1` This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost.

当 `acks` 设置为 1 时,生产者发送消息将等待这个分区的 Leader Server Broker 完成它本地

的消息记录操作，但不会等待这个分区下其他 Follower Server Brokers 的操作。在这种情况下，虽然 Leader Server Broker 对消息的处理成功了，也返回了 ACK 信息给生产者端，但是在进行副本复制时，还是可能失败。

acks=all (#注：官方原文如此，实际上属性值为“-1”) This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.

当 acks 设置为“all”时，消息生产者发送消息时将会等待目标分区的 Leader Server Broker 以及所有的 Follower Server Brokers 全部处理完，才会得到 ACK 确认信息。这样的处理逻辑下牺牲了一部分性能，但是消息存储可靠性是最高的。

## 9.6.2 Kafka 集群安装：配置过程

- 首先在 192.168.61.140 的服务器上安装 ZooKeeper 以后，直接启动 ZooKeeper 即可：

```
zkServer.sh start
```

- 可以在 Apache Kafka 的官方网站下载 V0.8.X 版本的安装包 (<http://kafka.apache.org/downloads.html>)，请不要下载 V0.9.X 版本的安装包，因为 V0.9.X 版本中消费者端的配置属性发生了相当的变化。本书的讲解将基于 V0.8.1.1 版本，并且全部针对 V0.8.X 版本兼容的配置属性：（[https://www.apache.org/dyn/closer.cgi?path=/kafka/0.8.1.1/kafka\\_2.10-0.8.1.1.tgz](https://www.apache.org/dyn/closer.cgi?path=/kafka/0.8.1.1/kafka_2.10-0.8.1.1.tgz)）。

可以直接使用 wget 命令，也可以通过浏览器（或者第三方软件）下载：

```
wget https://www.apache.org/dyn/closer.cgi?path=/kafka/0.8.1.1/kafka_2.10-0.8.1.1.tgz
```

- 下载后，运行命令进行压缩文件的解压操作：

```
tar -xvf ./kafka_2.10-0.8.1.1.tgz
```

笔者习惯将可运行软件放置在 /usr 目录下，读者可以按照自己的操作习惯或者所在团队的规范要求放置解压后的目录（正式环境下不建议使用 root 账号运行 Kafka）：

```
mv /root/kafka_2.10-0.8.1.1 /usr/kafka_2.10-0.8.1.1/
```

- Apache Kafka 所有的管理命令都存放在安装路径下的 /bin 目录中。所以，如果希望后续管理方便就可以设置一下环境变量：

```
export PATH=/usr/kafka_2.10-0.8.1.1/bin:$PATH
```

#记得在/etc/profile 文件的末尾加入相同的设置

- Apache Kafka 的配置文件存放在安装路径下的./config 目录下。如下所示:

```
-rw-rw-r--. 1 root root 1202 4月 22 2014 consumer.properties
-rw-rw-r--. 1 root root 3828 4月 22 2014 log4j.properties
-rw-rw-r--. 1 root root 2217 4月 22 2014 producer.properties
-rw-rw-r--. 1 root root 5322 4月 28 23:32 server.properties
-rw-rw-r--. 1 root root 3326 4月 22 2014 test-log4j.properties
-rw-rw-r--. 1 root root 995 4月 22 2014 tools-log4j.properties
-rw-rw-r--. 1 root root 1023 4月 22 2014 zookeeper.properties
```

如果进行的是 Apache Kafka 集群安装, 那么只需要关心 “server.properties” 这个配置文件。其中目录下有一个 zookeeper.properties 不建议使用。之所以有这个配置文件, 是因为 Kafka 中带有 ZooKeeper 运行环境, 如果使用 Kafka 中的 “zookeeper-server-start.sh” 命令启动这个自带 ZooKeeper 环境, 才会用到这个配置文件。

- 开始编辑 server.properties 配置文件。这个配置文件中默认的配置项就有很多, 但是不必全部进行更改。下面我们列举了更改后的配置文件情况, 其中主要需要关心的属性使用中文进行了说明 (当然原有的注释会进行保留):

```
# The id of the broker. This must be set to a unique integer for each
broker.
# 非常重要的一个属性, 在 Kafka 集群中每一个 Brocker 的 id 一定要不一样, 否则启动时会报错
broker.id=2
# The port the socket server listens on
port=9092
# Hostname the broker will bind to. If not set, the server will bind to
# all interfaces
#host.name=localhost
# The number of threads handling network requests
num.network.threads=2
# The number of threads doing disk I/O
# 顾名思义, 就是有多少个线程同时进行磁盘 I/O 操作
# 这个值实际上并不是设置得越大性能越好
# 例如提供给 Kafka 使用的文件系统物理层只有一个磁头在工作
# 那么这个值就变得没有任何意义了
num.io.threads=8
# The send buffer (SO_SNDBUF) used by the socket server
socket.send.buffer.bytes=1048576
# The receive buffer (SO_RCVBUF) used by the socket server
socket.receive.buffer.bytes=1048576
# The maximum size of a request that the socket server will accept
# (protection against OOM)
```



```

socket.request.max.bytes=104857600
# A comma seperated list of directories under which to store log files
# 很多开发人员在使用 Kafka 时，不重视这个属性
# 实际上 Kafka 的工作性能绝大部分就取决于你提供什么样的文件系统
log.dirs=/tmp/kafka-logs
# The default number of log partitions per topic. More partitions allow
# greater
# parallelism for consumption, but this will also result in more files
# across the brokers.
num.partitions=2
# The number of messages to accept before forcing a flush of data to
# disk
# 从 Page Cache 中将消息正式写入磁盘上的阈值：以待转储消息数量为依据
#log.flush.interval.messages=10000
# The maximum amount of time a message can sit in a log before we force
# a flush
# 从 Page Cache 中将消息正式写入磁盘上的阈值：以转储间隔时间为依据
#log.flush.interval.ms=1000
# The minimum age of a log file to be eligible for deletion
# log 消息信息保存时长，默认为 168 个小时
log.retention.hours=168
# A size-based retention policy for logs. Segments are pruned from the
# log as long as the remaining
# segments don't drop below log.retention.bytes.
# 默认为 1GB，在此之前 log 文件不会执行删除策略
# 实际环境中，由于磁盘空间根本不是问题，并且内存空间足够大。所以笔者会将这个值设置的较
# 大，例如 100GB。
#log.retention.bytes=1073741824
# The maximum size of a log segment file.
# When this size is reached a new log segment will be created.
# 默认为 512MB，当达到这个大小，Kafka 将为这个 Partition 创建一个新的分段文件
log.segment.bytes=536870912
# The interval at which log segments are checked to see if they can be
# deleted according
# to the retention policies
# 文件删除的保留策略，多久被检查一次（单位毫秒）
# 实际生产环境中，6~12 小时检查一次就够了
log.retention.check.interval.ms=60000
# By default the log cleaner is disabled and the log retention policy
# will default to just delete segments after their retention expires.
# If log.cleaner.enable=true is set the cleaner will be enabled and
# individual logs can then be marked for log compaction.
log.cleaner.enable=false
##### ZooKeeper #####

```



```
# ZooKeeper connection string (see zookeeper docs for details).
# root directory for all kafka znodes.
# 到 ZooKeeper 的连接信息, 如果有多个 ZooKeeper 服务节点, 则使用 “,” 进行分割
# 例如: 127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002
zookeeper.connect=192.168.61.140:2181
# Timeout in ms for connecting to zookeeper
# ZooKeeper 连接超时时间
zookeeper.connection.timeout.ms=1000000
```

以上系统自带的 **Broker** 服务节点的配置项还不是最完整的, 在官网 (<http://kafka.apache.org/documentation.html#brokerconfigs>) 上有完整的 “server.properties” 文件的配置属性和说明信息。

再次强调一下, 以上配置属性中必须按照自己的环境更改的属性有: “broker.id”、“log.dirs” 及 “zookeeper.connect”。其中每一个 **Kafka** 服务节点的 “broker.id” 属性都必须不一样。

- 这样我们就完成了其中一个 **Broker** 节点的安装和配置。接下来需要按照以上描述的步骤进行 **Kafka** 集群中另一个 **Broker** 节点的安装和配置。一定注意每一个 **Kafka** 服务节点的 “broker.id” 属性都必须不一样, 在本演示示例中, 笔者设置的 broker.id 分别为 1 和 2。
- 接下来启动 **Apache Kafka** 集群中已经完成安装和配置的两个 **Broker** 节点。如果以上所有步骤都正确完成了, 那么将会看到类似如下的启动日志输出:

```
# 分别在两个节点上执行这条命令, 以便完成节点启动:
kafka-server-start.sh /usr/kafka_2.10-0.8.1.1/config/server.properties
.....
[2016-04-30 02:53:17,787] INFO Awaiting socket connections on 0.0.0.0:9092.
(kafka.network.Acceptor)
[2016-04-30 02:53:17,799] INFO [Socket Server on Broker 2], Started
(kafka.network.SocketServer)
.....
```

启动成功后, 我们可以在某一个 **Kafka Broker** 节点上运行以下命令来创建一个 **Topic**。为了后续进行讲解, 我们创建的 **Topic** 有四个分区和两个复制因子:

```
kafka-topics.sh --create --zookeeper 192.168.61.139:2181
--replication-factor 2 --partitions 4 --topic my_topic2
```

### 9.6.3 Kafka 常用命令

在安装 **Kafka** 集群时, 使用到了 **Kafka** 提供的脚本命令进行集群启动、**Topic** 创建等相关操作。实际上 **Kafka** 提供了相当丰富的脚本命令, 以便于开发者进行集群管理、集群状态监控、消费者/生产者测试等工作, 这里为大家列举一些常用的命令:

## 1. 集群启动

```
kafka-server-start.sh config/server.properties
```

这个命令带有一个参数——指定启动服务所需要的配置文件。默认的配置文件中已经提到过，存在于 Kafka 安装路径的 `./config` 文件夹下，文件名为 `server.properties`。

## 2. 创建 Topic

```
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

带有 `-create` 参数的 `kafka-topics` 命令脚本用于在 Kafka 集群上创建一个新的 Topic。后续四个参数为：

- **ZooKeeper** 该参数用来指定 Kafka 集群所使用的 ZooKeeper 的地址，这是因为当 Topic 被创建时，ZooKeeper 下的 `/config/topics` 目录中会记录新的 Topic 的配置信息。
- **replication-factor** 复制因子数量。副本是 Kafka V0.8.X 版本中加入的保证消息可靠性的功能，复制因子是指某一条消息进行复制的副本数量，该功能以集群中 **Broker** 服务节点的数量为单位。也就是说当 Broker 服务节点的数量为 X 时，复制因子的数量最多为 X。否则在执行 Topic 创建时会报告类似如下的错误：

```
Error while executing topic command replication factor: 3 larger than available brokers: 2
```

- **Partitions** 分区数量（默认分区为 1）。一个 Topic 可以有若干分区，这些分区分布在 Kafka 集群的一个或者多个 Broker 上。**Partition** 分区是 Kafka 集群实现消息负载均衡功能的重要基础，且 Topic 中 **Partition** 分区一旦创建就不允许进行动态更改。所以一旦准备在正式生产环境创建 Topic，就一定要慎重考虑它的分区数量。
- Topic 新创建的 Topic 的名称。该参数在创建 Topic 时指定，且在 Kafka 集群中 Topic 的名称必须是唯一的。

## 3. 以生产者身份登录测试

```
kafka-console-producer.sh --broker-list localhost:9093 --topic test
# 或者
kafka-console-producer.sh --producer.config client-ssl.properties
```

使用命令脚本（而不是 Kafka 提供的各种语言的 API），模拟一个消息生产者登录集群，主要是为了测试指定的 Topic 的工作情况是否正常。可以有两种方式作为消息生产者登录 Kafka 集群：

第一种方式指定 `broker-list` 参数和 `Topic` 参数，`broker-list` 携带需要连接的一个或者多个 Broker 服务节点；Topic 为指定的该消息生产者所使用的 Topic 的名称。

第二种方式是指定 **Producer** 生产者配置文件和客户端 **ssl** 加密信息配置文件（如果没有在 **Kafka** 集群中配置 **ssl** 加密规则，后一个文件也可不进行指定）。默认的 **Producer** 生产者配置文件存放在 **Kafka** 安装路径的 `./config` 目录下，文件名为 `producer.properties`。

#### 4. 以消费者身份登录测试

```
kafka-console-consumer.sh --zookeeper localhost:2181 --topic test
```

同样可以使用命令脚本的方式，以消息消费者的身份登录 **Kafka** 集群，目的相同：为了测试 **Kafka** 集群下创建的 **Topic** 是否能够正常工作。该命令有两个参数：

- **ZooKeeper** 指定的 **Kafka** 集群所使用的 **ZooKeeper** 地址，如果有多个 **ZooKeeper** 节点就使用“,”进行分割。该参数必须进行指定。
- **Topic** 该参数用于指定使用的 **Topic** 名称信息。如果 **Topic** 在 **Kafka** 集群下工作正常，那么在成功使用消费者身份登录后，就可以收到 **Topic** 中有生产者发送的消息信息了。

#### 5. 查看 Topic 状态

```
kafka-topics.sh --describe --zookeeper 192.168.61.139:2181 --topic my_topic2
```

以上命令可以用来查询指定的 **Topic** (`my_topic2`) 的关键属性，包括 **Topic** 的名称、分区情况、每个分区的主控节点、复制因子、复制序列已经赋值序列的同步状态等信息。命令可能的结果如下所示：

```
Topic:my_topic2 PartitionCount:4      ReplicationFactor:2       Configs:
Topic: my_topic2  Partition: 0  Leader: 2  Replicas: 2,1  Isr: 2,1
Topic: my_topic2  Partition: 1  Leader: 1  Replicas: 1,2  Isr: 1,2
Topic: my_topic2  Partition: 2  Leader: 2  Replicas: 2,1  Isr: 2,1
Topic: my_topic2  Partition: 3  Leader: 1  Replicas: 1,2  Isr: 1,2
```

以上命令，返回结果的意义已经讲过了，可以参见 9.6.1 节中关于 **Kafka** 消息复制功能的描述部分，这里就不再进行赘述了。

## 第四部分

# 场景实战

本部分列举了两个实战场景，通过介绍这两个实战场景，本书将和读者一起把讲解过的知识综合起来应用到实际开发过程中。这两个实战场景一个是收集用户操作事件的日志采集系统，一个是可用于大规模图片处理的图片管理服务。我们将在其中使用负载均衡技术、存储技术、消息队列技术、系统间服务调用技术及图片处理技术等。对于一些本书之前没有详细讲解到，但又必须要使用的技术知识点，本部分内容也做了概括性讲解，例如磁盘阵列系统和采集日志所需要的 Apache Flume，等等。

# 第 10 章

## 场景实战：其他储备知识

### 10.1 数据存储

由于本书篇幅的原因，数据存储相关的知识点虽然没有作为本书的重点，但是为了便于读者更好理解本章场景实践中的示例，本小节需要为各位读者简单介绍一些关于系统存储的知识。首先应该明白数据存储的重要性：在进行业务计算前的初始化数据、计算过程中的临时数据、计算完成后得到的计算结果都需要进行存储。我们可以通过一张示例图首先概述一下存储技术的分层思路（图 10-1）。

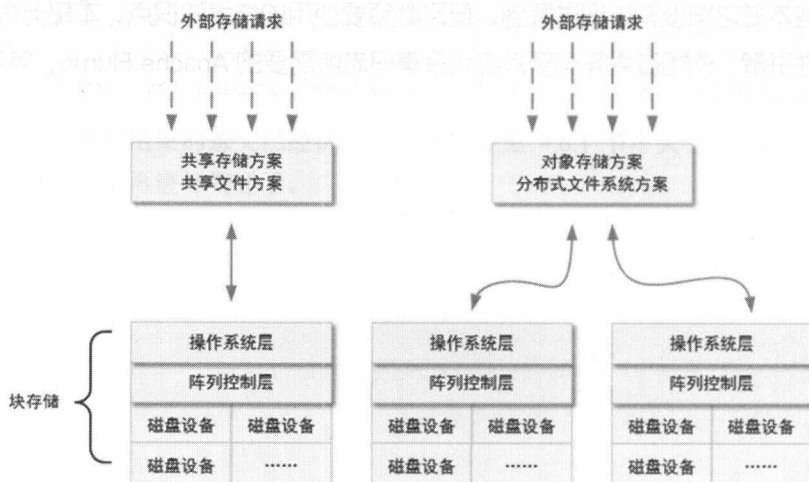


图 10-1 存储体系



### 10.1.1 块存储

磁盘设备就是我们熟悉的硬盘，目前最常见的就是拥有各种数据接口的机械硬盘，还有固态硬盘。而且随着后者价格越来越便宜，在不久的将来肯定会出现固态硬盘全面替代机械硬盘的情况。我们来回忆一下在 CentOS 6.X 操作系统上，如何进行一个物理磁盘的操作。

- 首先通过 `fdisk` 命令对本地硬盘进行分区（即确定可控制的扇区的范围）。在这个过程中，操作者实际上无须关心被分区的磁盘设备到底是机械硬盘还是固态硬盘又或者是一个磁盘阵列，在操作系统中它的表现都是一个设备文件，如图 10-2 所示。

```
[root@localhost yinwenjie]# fdisk /dev/sdb
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel with disk identifier 0x2d5a20b1.
Changes will remain in memory only, until you decide to write them.
After that, of course, the previous content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)

WARNING: DOS-compatible mode is deprecated. It's strongly recommended to
switch off the mode (command 'c') and change display units to
sectors (command 'u').

Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1305, default 1):
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-1305, default 1305):
Using default value 1305

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
```

图 10-2 `fdisk` 命令

- 接下来，操作人员会在这个分区上通过 `mkfs` 命令创建想要的文件系统（Ext3、Ext4、LVM、XFS、BTRFS 等）。这些文件系统实际上规定了数据在硬件上的存取规则，如图 10-3 所示。

```
[root@localhost yinwenjie]# mkfs.ext4 /dev/sdb1
mke2fs 1.41.12 (17-May-2010)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
655360 inodes, 2620595 blocks
131029 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=2684354560
80 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

图 10-3 mkfs 命令

以上演示的过程实际上就是一个最简单的块操作过程：这些文件系统是操作系统对下层存储设备进行操作的关键。根据不同格式的文件系统，操作系统将这些物理设备划分为不同的块，一个块可能包含底层机械磁盘的若干个连续扇区（称之为簇）。这样一来作为操作系统的使用者，就不再需要关心底层设备的扇面、扇区、磁道这些概念了：

- 物理块，一个物理块是上层文件系统能够操作的最小单位（通常为 512 字节），一个物理块在设备底层对应了多个连续的物理扇区。通常一块 SATA 硬盘会有若干机械手臂（决定于物理盘片数量），和若干个物理扇区。物理扇区的大小是磁盘出厂时就确定的，我们无法改变。
- 单个扇区的工作是单向的，那么映射出来的一个物理块的工作方式也是单向的。原理就是机械手臂在读取这个扇区的数据时，硬件芯片是不允许机械手臂同时向这个扇区写入数据的。
- 通过上层文件系统（EXT、NTFS、BTRFS、XFS）对下层物理块的封装，操作系统就不需要直接操作磁盘物理块了。操作者通过 ls 这样的命令看到的一个一个文件也不需要关心这些文件在物理块的存储格式。另外，不同的文件系统有不同的功能特性（有的文件系统支持快照，有的文件系统支持数据恢复），基本原理就是这些文件系统对下层物理块的操作规范不一样。

### 10.1.2 共享存储/共享文件存储

10.1.1 节已经提到块存储是不能进行设备共享的，最根本的原因是底层设备不允许机械手臂同时向一个扇区进行写操作，操作系统层面上文件系统的块索引记录也只能存放在操作系统本地。你见过两个不同的操作系统同时向一个机械磁盘设备写入文件吗？

那么如何做到不同的操作系统同时向一个硬件设备写入文件呢？显然在块存储上面做文章不现实，因为其定义已经决定了不可能，那么最直接的办法就是在块存储方案之上再进行一次封装，如图 10-4 所示。

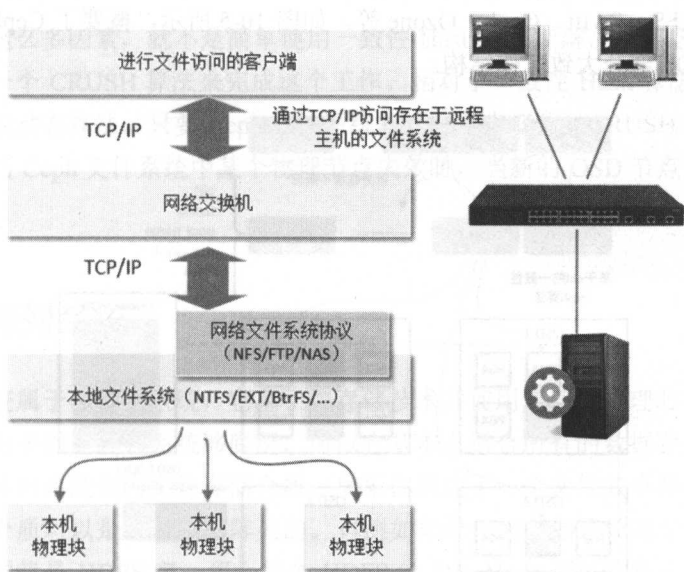


图 10-4 共享存储

- NFS、FTP 这些技术名词是不是很熟悉？文件共享系统的关键在于，文件共享系统不再请求文件数据读写操作的本地计算机，而是通过网络访问存在于远程的控制系统，再由远程的文件系统操作块 I/O 命令完成数据操作。
- 诸如本地文件系统 NTFS/EXT/LVM/XFS 等是不允许直接进行网络访问的，所以一般文件存储系统会进行一层网络协议封装，这就是 NFS 协议/FTP 协议/NAS 协议（注意，这里说的是协议），再由协议操作文件存储系统的服务器文件系统。而保证多个操作系统同时进行磁盘设备的读写操作，也是由 NFS/FTP/NAS 这些协议提供的。
- 文件存储系统要解决的首要问题是文件共享，网络文件协议可以保证多台客户端共享服务器上的文件结构。从整个架构图上可以看到，文件存储系统的数据读写速度、数据吞吐量是没办法和块存储系统相比的（因为这不是文件共享系统要解决的首要问题）。

当面对大量的数据读写压力时，共享存储系统肯定不是首要选择，而当需要选择块存储系统时又要面临成本和运维的双重压力（SAN 系统的搭建比较复杂，而且建设费用昂贵）。如果在实际生产环境中我们经常遇到数据读取压力大，且需要共享文件信息的场景，那么这个问题怎么解决呢？

### 10.1.3 对象存储系统

对象存储系统兼具块存储系统的高吞吐量、高稳定性和文件存储的网络共享性、廉价性的特点。可以说对象存储系统就是为了满足 10.1.2 节末尾我们提到的需求而出现的。典型的对象存储系统包括：MFS、Swift、Ceph、Ozone 等。如图 10-5 所示，概要了 Ceph 这款对象存储系统最核心的 RADOS 部分大致工作结构。

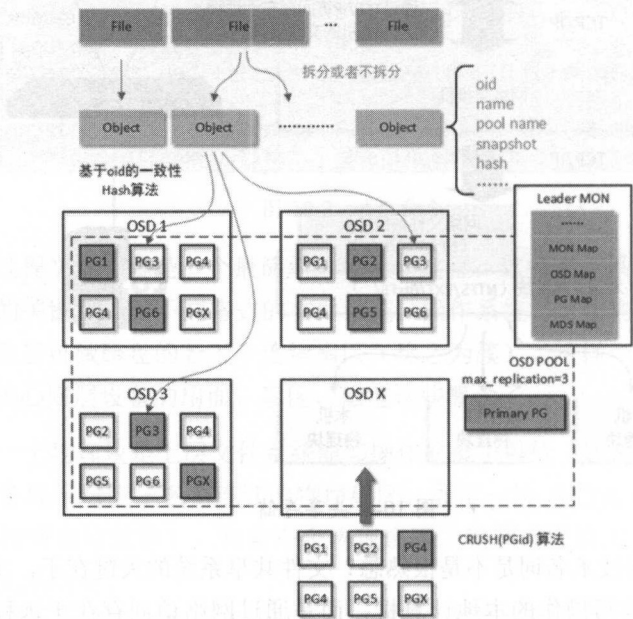


图 10-5 Ceph 中的 RADOS

首先 RADOS 会将这个文件转换成一个或者多个 Object。每个 Object 的最大大小是有限制的（默认为 4MB），所以如果文件太大，就会拆分为多个 Object。每个 Object 信息中有几个重要属性，例如 oid 表示这个 Object 在 Ceph 文件系统中的唯一编号，name 表示 Object 的唯一名字、snapshot 快照信息和已确定的所在 pool 的对应关系，等等。

转换为 Object 就可以开始进行真正的存储了，存储 Object 信息的单元称为 PG（Placement Group），每个 PG 可以存储大量 Object 信息，并且 PG 还可以分为 Primary PG 和 Replicas PG。

例如图 10-5 编号为 PG1 的 PG 块就有三个，分别存放在三个不同的 OSD 角色中，其中灰色底的 PG1 是 Primary PG，另外两个是 Replicas PG。也就是说这个 OSD Pool 中的 PG 副本数量为 3。

Ceph 文件系统中最终落地存储的角色是 OSD（对象存储设备），也就是说在确定 Object 和 PG 的对应关系后，还需要确定 PG 和 OSD 的对应关系，这个过程所需要考虑的因素就比在确认 Object 和 OSD 的对应关系时考虑的因素多得多了，例如承载 OSD 角色的物理节点其性能可能不一样，在确定承载 PG 的 OSD 时不仅要考虑散列度，还需要考虑节点的性能问题；再例如多个 OSD 角色可能存在于同一个物理节点上，那么相同 PG 的 Primary PG 和 Replicas PG 最好就不要放在同一个物理节点上，以免当物理节点崩溃后，相同 PG 的多个副本一起失效。

显然要考虑这么多因素，就不是简单使用一致性 Hash 算法能解决的问题了。所以 Ceph 文件系统中专门有一个 CRUSH 算法来完成这个工作。相对于一致性 Hash 算法，CRUSH 算法是考虑的多种因素的动态算法，只要 Ceph 文件系统的状态发生变化，CRUSH 算法的计算结果就会不一样。例如当 Ceph 文件系统中某个物理节点失效时，当新的 OSD 节点加入并开始承载新的 PG 时。

## 10.2 磁盘阵列系统

磁盘阵列系统属于硬件层系统，它基于块存储技术并使用专门的管理芯片对多块物理磁盘进行读写管理。由于磁盘阵列系统够底层，所以它基本上适合所有的数据存储场景。例如你可以直接使用磁盘阵列系统作为文件存储介质，如果你搭建了一个文件共享存储方案，那么这个共享存储的物理介质可以是基于磁盘阵列的。再例如如果你的存储系统基于 20 个节点的分布式文件系统，比如就是 HDFS 吧，那么每个 HDFS 的 DataNode 都可以基于一个微型的磁盘阵列系统。要知道单块磁盘进行数据存储可能会存在以下问题：

- 磁盘容量有限制，当容量不足时不能进行硬件扩容。现在磁盘技术在磁盘容量上已经有了长足的发展，目前（2017 年）机械硬盘的主流容量已经达到 6TB，固态硬盘的主流容量也达到 512GB。但是单块硬盘始终都存在较严重的容量扩充问题，除非读者在扩容时手动迁移数据。
- 数据可靠性问题。单块硬盘不存在任何备份机制，虽然现在有很多扇区检测软件可以帮助开发人员/运维人员提前发现硬盘损坏的磁道，但这都不能保证 99.99% 的运行可靠性。一旦硬盘由于各种原因损坏（电压不稳、磁头位移等），存储在其上的数据就可能永久丢失。
- 读写性能瓶颈。这个问题在 SSD 固态硬盘上还不太明显，目前主流的固态硬盘的外部传输速度可达到 550MB/s，这个速度基本上达到了 SATA 3/USB 3.0 接口规范的理论峰



值。但这个问题对于机械硬盘来说却很明显了，由于机械硬盘的读写性能受到磁头数量、盘片转速、盘片工艺等因素的影响，所以机械硬盘的读写性能一直没有一个质的飞跃。如果将单个硬盘应用在生产系统上，那么磁盘读写性能无疑将会是整个系统的性能瓶颈。另外 SATA 3 结构理论上 6Gbps 的传输带宽必要时也需要找到替代方案。

为了解决以上这些问题，硬件工程师将多个硬盘按照不同的规则组合在一起形成各种集群化的数据存储结构，这些存储结构被称为磁盘阵列（Redundant Arrays of Independent Disks，RAID）。磁盘阵列解决以上这些问题的基本思路有：

- 通过硬盘横向扩展或者纵向扩展的方式，解决整个磁盘整理存储容量限制的问题。而对于上层操作系统来说，看到的都只是一个磁盘设备文件/操作盘符而已。
- 通过数据镜像或者数据校验的方式创建数据冗余，从而解决数据恢复问题。
- 通过阵列控制芯片分发数据读写请求的方式，将原本集中在一块硬盘上的数据读写请求分散到多块硬盘上，从而解决磁盘性能的问题。

### 10.2.1 RAID 0

RAID 0 阵列结构是所有阵列结构中读写性能最好的，也是所有阵列结构中实现思路最简单的，如图 10-6 所示。

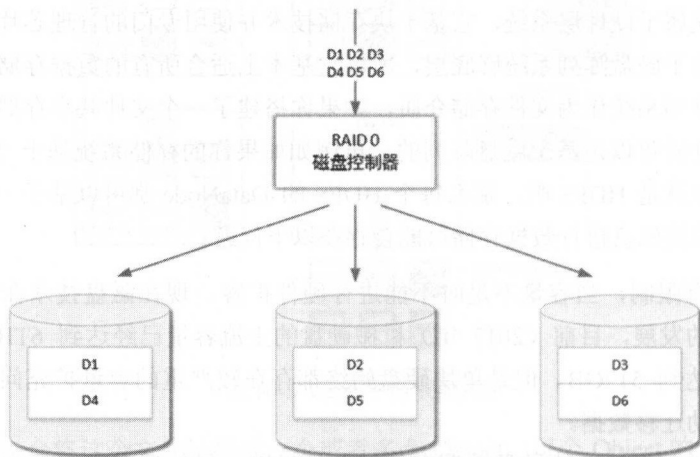


图 10-6 RAID 0

RAID 0 阵列结构没有数据冗余机制和数据恢复机制，它至少需要两个硬盘进行构造。整个 RAID 0 阵列结构就是将参与 RAID 0 阵列构建的所有硬盘进行容量累加，从而形成一个更大、对上层操作系统统一的存储容量。所以，RAID 0 阵列的存储容量就是这些硬盘的容量累加。

当需要写入的数据到达阵列控制器，后者会向其下的硬盘设备分发这些数据。这样原来只由一个硬盘承担的读写压力就会被分摊到多个硬盘上，最终提高了整个阵列的读写性能。RAID 0 阵列结构存储速度的优势非常明显，且参与构造阵列的磁盘数量越多阵列速度越快（峰值速度最终会受到总线、外部接口规范、控制芯片制造工艺等因素的限制）。但是 RAID 0 阵列结构的缺点也很明显：由于阵列结构没有容错机制或者数据恢复机制，当阵列中的一个或者多个磁盘发生故障时，整个阵列结构就会崩溃并且不能恢复。所以在实际应用中，只有那些单位价值不高且每天又需要大量存储的数据才会使用 RAID 0 阵列结构进行存储，例如日志文件数据。

### 10.2.2 RAID 1

RAID 1 阵列结构又被称为磁盘镜像阵列或者磁盘冗余阵列。它的构造特点是阵列结构中的每一个磁盘互为镜像，如图 10-7 所示。

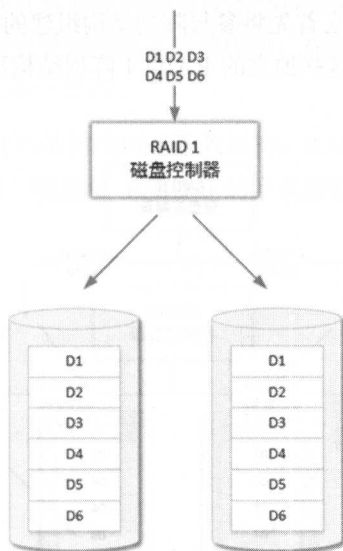


图 10-7 RAID 1

当有外部数据需要存储时，RAID 1 阵列控制器将首先把这个数据做成  $N$  个副本（ $N$  的数量和阵列结构中物理磁盘的数量相等），实际上镜像副本的单位为扇区或者 Flash Page。这些副本会分别存储到阵列结构的各个磁盘中。在进行数据读取时，RAID 1 阵列结构中的某一块磁盘将会作为主要的数据读取源头，当这个源头出现吞吐量瓶颈时，RAID 1 阵列控制器会主动到其他镜像磁盘读取数据。所以 RAID 1 阵列的数据读取性能还是要比单个磁盘的性能要好，但是写入性能却差了很多。

从以上介绍可以看出，RAID 1 阵列结构设计之初的主要目的并不是提高存储设备的读写性能，而是保证高价值数据的存储可靠性。由于 RAID 1 阵列结构中需要保证每个磁盘的镜像数据完全一致，所以它还要求参与 RAID 1 阵列结构的每一个磁盘的容量必须相同，否则 RAID 1 阵列结构会以最小的那个磁盘容量为自己的标准容量。

### 10.2.3 RAID 10 和 RAID 01

RAID 0 和 RAID 1 都有自己的优缺点，并且优点都很突出：RAID 0 虽然速度快但是没有任何数据保障措施，所以一味地快意义并不大；RAID 1 虽然保证了数据的可靠性，但是却牺牲了大量空间和读写速度。所以以上两种阵列结构特别是 RAID 0，在企业级/工业级环境中使用的情况还是比较少。那么有没有一种阵列结构在融合了 RAID 0 和 RAID 1 两者优点的同时又避免了各自的缺点呢？

肯定是有的，RAID 10 和 RAID 01 两种阵列结构就是为了实现 RAID 0 和 RAID 1 的融合而被设计的。在 RAID 10 结构中，它首先将参与阵列结构组建的磁盘进行分组，形成若干组独立的 RAID 1 阵列结构，然后再将这些独立的 RAID 1 阵列结构形成 RAID 0 结构，如图 10-8 所示。

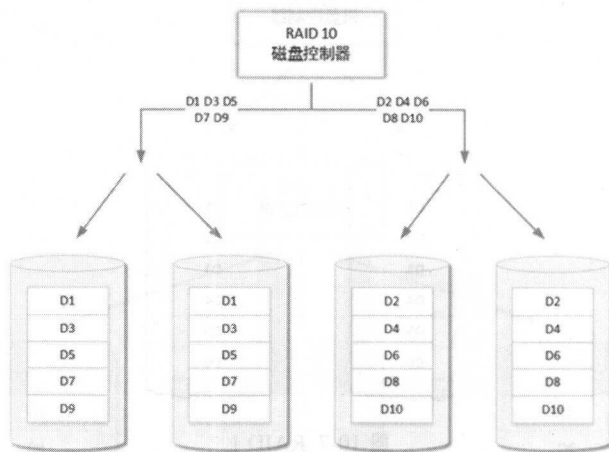


图 10-8 RAID 10

图 10-8 中有四块硬盘参与 RAID 10 阵列结构的组建，四块硬盘是组建 RAID 10 阵列结构的最小要求（实际上两块也行，但是那样的 RAID 10 没有任何意义）。它们首先被两两分组形成两个独立的 RAID 1 结构，这也意味着这些硬盘的容量最好是一样的，否则每组 RAID 1 结构会基于容量最小的那块硬盘确认自己的容量。接着独立工作的两组 RAID 1 再组成 RAID 0 阵列结构。

假设参与 RAID 10 构建的硬盘大小都为 6TB，则两组独立的 RAID 1 阵列结构的容量分别为 12TB，最终整个 RAID 10 阵列结构的存储容量为 12TB。可以看到 RAID10 阵列结构的存储容量和独立磁盘的大小、分组数量有关。我们可以得到以下的计算公式：

RAID 10 总容量 =  $N / G \times$  单个硬盘的存储容量

这个公式假设的前提是参与 RAID 10 构建的每个硬盘的存储容量都相同。其中  $N$  表示参与 RAID 10 构建的硬盘总数， $G$  代表 RAID 10 下磁盘映射的分组数量（RAID 1 分组数量）。例如，总共 12 块硬盘参与 RAID 10 构建，每个硬盘的大小为 6TB，且分为三组 RAID 1，那么这样组建的 RAID 10 阵列结构的存储容量为 24TB；如果同样的情况下，这些硬盘被分为四组 RAID 1，那么组建的 RAID 10 阵列结构的存储容量就为 18TB。

可见 RAID 10 通过集成更多硬盘的思路，将 RAID 0 阵列和 RAID 1 阵列的特点进行了融合，在保证数据存储可靠性的基础上提高了阵列的整体存储性能。RAID 10 被广泛应用在各种计算场景中，市场上从几千元到几百万元的阵列设备都提供对 RAID 10 磁盘阵列结构的支持。RAID 10 磁盘阵列的总读写速度会受到控制芯片的影响，所以售价几千元和售价上百万元的磁盘阵列设备实际读写性能是完全不一样的。

另外还有一种和 RAID 10 阵列结构相似的阵列结构：RAID 01（或称为 RAID 0 + 1），它们的构造区别是，后者首先将若干磁盘以 RAID 0 的方式进行组织，然后再分组成多个独立的 RAID 1 结构，如图 10-9 所示。

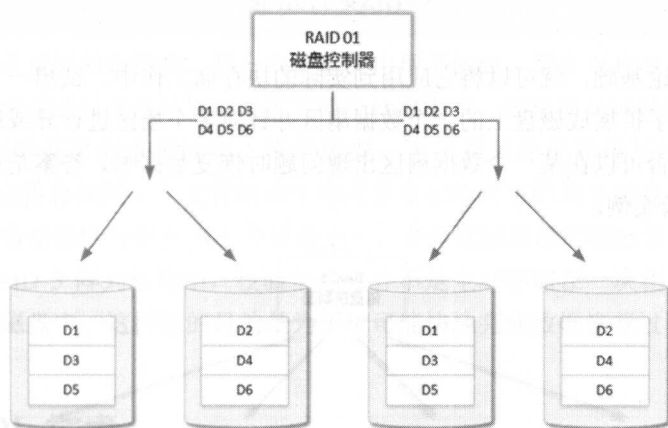


图 10-9 RAID 01

## 10.2.4 RAID 5

虽然速度上 RAID 5 没有 RAID 10/01 阵列结构快，但是 RAID 5 阵列控制芯片的成本却低

很多。RAID 5 阵列基于奇偶校验原理，它的算法核心是异或运算（XOR）。异或运算是各位读者在大学离散数学课程中学习过的一种基本二进制运算，其运算关系如表 10-1 所述（表 10-1 的计算因子只有两个，目的是让读者回忆起来）。

表 10-1

A 值	B 值	运算结果
1	1	0
1	0	1
0	1	1
0	0	0

接着我们可以再假设计算因子为  $N$ ，根据异或运算的特点，可以在已知结果和  $N-1$  个原始计算因子的前提下，还原出未知的那个计算因子。请看下面示例的计算过程（ $N=4$ ）：

.....

$1 \wedge 1 \wedge 1 \wedge ? = 1$

$1 \wedge 1 \wedge 1 \wedge ? = 0$

$1 \wedge 0 \wedge 1 \wedge ? = 1$

$1 \wedge 0 \wedge 1 \wedge ? = 0$

$0 \wedge 0 \wedge 1 \wedge ? = 1$

$0 \wedge 0 \wedge 1 \wedge ? = 0$

$0 \wedge 0 \wedge 0 \wedge ? = 1$

$0 \wedge 0 \wedge 0 \wedge ? = 0$

----->

----->

----->

----->

----->

----->

----->

----->

$? = 0$

$? = 1$

$? = 1$

$? = 0$

$? = 0$

$? = 1$

$? = 1$

$? = 0$

.....

有了以上的理论基础，就可以将它应用到实际的块存储工作中。试想一下如果将以上异或运算的每个计算因子扩展成磁盘上的一个数据扇区并针对多个扇区进行异或运算并将计算结果存储下来，那么是否可以在某一个数据扇区出现问题时恢复数据呢？答案是肯定的，请看如图 10-10 所示扇区校验实例。

The diagram illustrates the RAID 5 storage architecture. At the top, a box labeled "RAID 5 磁盘控制器" (RAID 5 Disk Controller) is connected by arrows to four data sectors and one parity sector. The data sectors are labeled "扇区A1 数据信息 512字节" (Sector A1 Data Information 512 bytes), "扇区A2 数据信息 512字节" (Sector A2 Data Information 512 bytes), "扇区A3 数据信息 512字节" (Sector A3 Data Information 512 bytes), and "扇区A4 校验信息 512字节" (Sector A4 Parity Information 512 bytes). The parity sector contains the binary value "10010101011110".

图 10-10 RAID 5 存储实例

• 350 •



在以上四个扇区的校验示例中，它们分属四个不同的磁盘设备，其中三个扇区存储的是数据，最后一个扇区存储的是异或运算后的校验码。在 10.1.1 节中我们已经介绍过一个扇区存储的数据量为 512 字节。当某个数据扇区出现故障时，基于校验扇区的信息和正常状态的数据扇区的信息，RAID 5 磁盘阵列可以将发生故障的扇区恢复出来；当某个校验扇区的信息出现故障时，RAID 5 磁盘阵列还可以重新进行校验。也就是说 RAID 5 阵列结构同一时间内只允许有一块硬盘出现故障，出现故障的硬盘需要立即进行更换（图 10-11）。

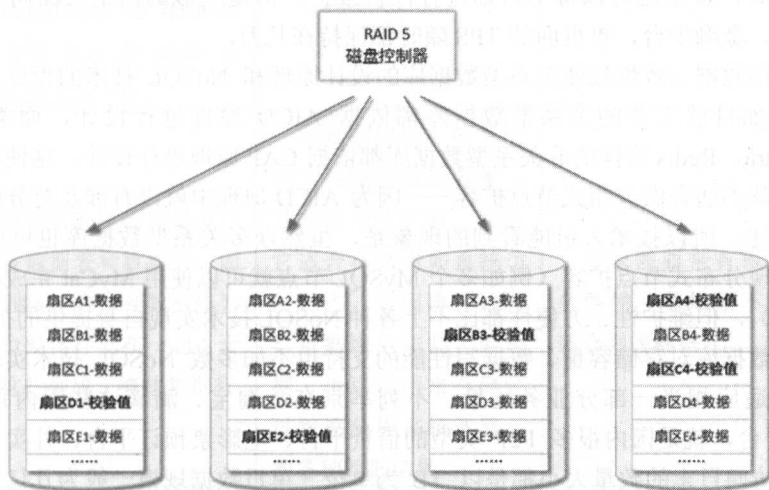


图 10-11 RAID 5

如果还未来得及更换故障硬盘，另一块硬盘又出现了故障，那么对整个 RAID 5 阵列就是毁灭性的——因为无法通过异或计算同时恢复两个计算因子。当更换故障硬盘后，RAID 5 阵列控制器将会自动对数据进行重新校验，恢复数据。为了在可靠性和读写性能上找到平衡，RAID 5 阵列结构会将存储同一个文件的若干扇区分布在阵列下的若干磁盘上（设阵列中磁盘总数为  $N$ ，则文件数据扇区分布于  $N-1$  个磁盘上），并将这些扇区的校验信息存储在最后剩余的一块磁盘上；RAID 5 阵列结构中，校验信息也并不是全部存储在一块相同的磁盘上，而是均匀分布在每一块磁盘中，这样做的目的是为了尽可能快地完成数据恢复过程。

## 10.3 NoSQL 技术

NoSQL 技术全称 Not Only SQL，又被称为非关系型数据库技术，它是一系列存储技术的总称。在当前移动互联网、云计算、分布式、微服务大行其道的环境下，技术人员渐渐发现传统的关系型数据库往往成为整个系统的瓶颈，引起这个问题的原因在本书 8.4.2 节介绍事务补

偿机制时就已进行过说明：传统的关系型数据库是一种强一致性系统，为了保证达到这个目的，系统的设计往往会牺牲分区隔离性和可用性，就连支撑关系型数据库构造分布式环境的分布式事务机制也特别突出数据一致性：

- 保证数据一致性在 10 年前的企业应用环境下来看，并没有任何问题。因为这些业务系统的主要要求就是要保证非常高的数据一致性，对于并发性能并没有太多要求，单页面的 TPS 峰值能达到几千就已经是一个较大的企业级应用了，这些系统对于可用性也要求不高，甚至还可以每个月都进行停机维护。但是类似国内主要面向终端用户的电商平台、金融平台，单页面的 TPS 随时都保持在几万。
- 过度强调数据一致性致使关系型数据库的设计原理和 NoSQL 技术的设计原理完全不一样。例如目前主要的关系型数据库都依据 AICD 原理进行设计，而类似 HBase、Cassandra、Redis 这样的非关系型数据库都依据 CAP 原理进行设计。这使得关系型数据库本身就不适合做分布式节点扩容——因为 AICD 原理中就没有涉及对分区容忍度的导向性描述。所以技术人员能看到的现象是，虽然众多关系型数据库也可以利用第三方组件实现分布式节点扩容（例如多个 MySQL 节点就可以使用 MyCat 完成分库分表和事务控制），但维护性、方便性都比不上各种 NoSQL 技术实现自身提供的扩容方案。
- 关系型数据库对存储容量、数据写性能的支持也不如多数 NoSQL 技术实现，所以前者也不再适应相当一部分业务场景。不列举京东、淘宝、滴滴这些国内顶级的电商、SaaS 平台，就是国内很多 P2P 类型的借贷平台、电影票预订平台、外卖平台，每天的 Nginx 访问日志的数量大小都是以 TB 为单位（单日数据规模一般为几亿条），而且对数据层的写并发性要求非常高。这些数据一般都会进行全量采集和存储，以便作为数据分析的基本依据。那么关系型数据库如何存储这些数据呢？不可能一条访问日志都按照数据库中一条记录进行存储吧。如果真是这样，每秒上万行且持续 10 多个小时的写操作问题，单单依靠关系型数据库又怎么解决呢？

目前业界流行的 NoSQL 技术总的来说可以分为四大类：列簇数据库、键值数据库、文档数据库和图数据库。其中前三类非关系型数据库技术，是我们在日常生产环境中常会使用到的。例如我们常常使用的 Redis 属于典型的键值数据库，而 MongoDB 就是一种文档型数据库。表 10-2 来源于网络并由笔者归纳整理，描述了目前业界流行的四类 NoSQL 数据库，包括它们的描述定义、适用和不适用的业务场景以及典型软件举例。

表 10-2

分类	数据模型	典型应用场景	不适用场景	软件举例
键 值 数据库	键值数据库是最简单的 NoSQL 数据库。其主要特点就是按照 Key—Value 的形式存储数据，	键值数据类一般用于缓存层，例如可以使用 Redis 存储当前已登录的用户信息、正在被多个	也是由于它们内部结构简单，所以它们不适合存储那些有复杂关联的数据。	Redis Memcached LevelDB

续表

分类	数据模型	典型应用场景	不适用场景	软件举例
	而 Key 一般使用各种 Hash 算法计算出一个唯一的数值依据。例如 Redis 中就采用一种名为 MurmurHash2 的 Hash 算法计算 Key 的 Hash 值（且不止这一种）	用户访问的页面数据，等等。这类数据库典型的代表有 Redis		
列 簇 数据库	列簇数据库的存储单位不是数据表和数据行，而是列簇。将一系列数据作为一个整体存储在一起，无论这些列属于哪些数据行	列簇数据库一般具有较大规模的数据容量，且非常方便扩容。那么它适合存储诸如系统访问日志、访问事件这样日数据增量非常大的数据，也可以用来存储经过数据分析系统计算后的各种数据结果	常常需要带有事务性的业务场景不适合列簇数据库	Cassandra HBase
文 档 数据库	这类数据库可存放并获取文档，例如 XML、JSON 格式的文档。其共同特点是带有一定描述结构，例如 XML、JSON 格式的文档本身就是按照树形结构进行组织的	这类数据库一般也能够存储大规模的数据量，且在保持了数据弱一致性的基础上提供了很好的数据库写性能。所以这类数据库适合存储应用程序的日志信息、具有信息结构内聚性评论信息、用户基本信息	文档数据库存储的“有结构”的信息，需要具有两个特点：第一是内聚性，也就是说这些使用这些数据的其他数据不需要知道前者的数据组织结构；第二是不变性，即是说数据结构本身是稳定的，不会经常出现结构变化	MongoDB CouchDB
图 数 据库	图数据库中存储单位是“节点”，存储的是“节点”的基本数据以及“节	现在的移动互联系统中有大量的事物关系场景，例如社交系统的一	和 其 他 几 类 NoSQL 数据库类似，图数据库也不	Neo4J

续表

分类	数据模型	典型应用场景	不适用场景	软件举例
	点”与“节点”之间的关联关系	度人脉和二度人脉，再例如用户推荐关系。这些数据如果使用图数据进行存储和查询，则可以大大简化应用程序层的编码复杂度	适合应用在常常需要带有事务性的业务场景中	

在本书场景实战部分，我们解决大规模数据存储的主要方法就是利用各种非关系型数据库的实现组件，本书将默认读者已经理解和掌握了 Redis 的基本使用方法，当然要是读者本身就可以独立搭建 Redis 高可用、高性能集群那就再好不过了。如果部分读者还需要重温以下 Redis 的重要知识点，还可以参考以下几篇笔者另外发布的博客资源：

《架构设计：系统存储（15）——Redis 基本概念和安装使用》

<http://blog.csdn.net/yinwenjie/article/details/53407288>

《架构设计：系统存储（16）——Redis 事件订阅和持久化存储》

<http://blog.csdn.net/yinwenjie/article/details/53518286>

《架构设计：系统存储（17）——Redis 集群方案：高可用》

<http://blog.csdn.net/yinwenjie/article/details/53672232>

《架构设计：系统存储（18）——Redis 集群方案：高性能》

<http://blog.csdn.net/yinwenjie/article/details/53905637>

# 第 11 章

## 场景实战：Kafka 与日志采集

### 11.1 Kafka 应用场景：场景说明

事件/日志收集是大中型软件不得不面对的话题。专门用于向其他业务系统提供事件/日志收集的系統，在本书被称为“事件/日志收集”系统。目前业务子系统对“事件/日志效果”系统功能的集成思路主要有两大类：**侵入式收集方案**和**非侵入式收集方案**，从数据分析便利性的角度来分类也分为埋点式和非埋点式。侵入式收集方案，是指任何需要使用事件/日志收集功能的业务系统，都需要做有针对性的编码工作，这个编码工作或者是新增代码用于调用“事件/日志收集”系统提供的客户端 API，又或者是修改已有的代码，以便适应事件/日志收集系统的调用特性。

侵入式方案又分为半侵入式和全侵入式。由于业务系统的代码结构本身存在问题，或者日益积累的代码腐化度，导致一旦需要集成“事件/日志收集”系统（或者任何其他第三方系统），就会出现需要调整业务系统中业务代码的现象。或者是由于需求变动导致的业务代码变动，也会牵扯到业务系统中集成的任意第三方系统的代码改变。这样的集成方式就是全侵入式的：出现这种情况一般是业务系统所选择的技术方案和业务系统本身的工程结构问题共同造成的。

很显然，全侵入式的方案是一种不佳的设计实践，在日常的设计工作中要尽量避免。半侵入式方案比全侵入方案要好很多：虽然业务系统会针对“事件/日志收集”系统的集成做一定的代码改造，但是由于业务系统的软件架构层次清晰，所以这部分代码和系统中原有的业务代码是完全分离的，只需要改造一次就可一直使用。也不会对业务系统既有的业务处理过程产生任何影响。或者是，业务系统由于需求变化产生的业务代码变化也不会造成“事件/日志收



集”系统的集成代码变化。

“事件/日志收集”系统的另外一种设计方案是非侵入式的。即业务系统在集成“事件/日志收集”系统时，不需要为这件事情专门引入新的代码或者修改已有代码。业务系统的开发人员甚至完全不知道（也不必知道）自己的系统集成成了“事件/日志收集系统”，仅通过配置一些参数文件的方式就可完成集成工作。

从本章开始，本节的最后一部分内容将利用已经介绍过的技术知识向大家讨论“事件/日志收集系统”的半侵入方案和非侵入式方案。当然中间还会穿插一些新技术的介绍，比如 Apache Flume。首先我们需要介绍一下示例中的业务场景，如图 11-1 所示。

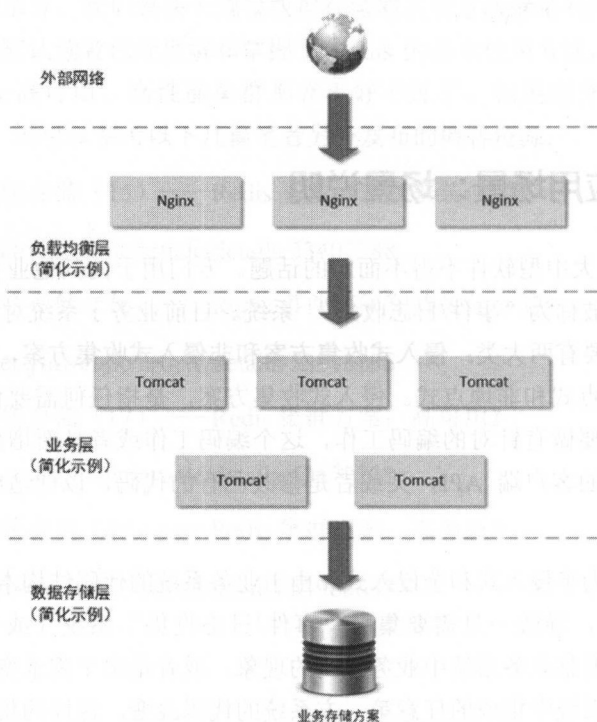


图 11-1 日志采集系统的业务场景

这是一个日均 200 万 PV 的中型电商网站的一个系统模块：商品详情模块。这个模块用于（且只用于）向用户展示商品详情、商品价格走势。图 11-1 中只列举了该模块使用的主要技术组件，毕竟这个实例场景不是为了讨论这些技术选型本身。由于网站业务的发展需要，需要在这个模块加入用户操作的统计分析功能，对用户“点击查看订单详情”、“点击查看商品价格走势”等操作动作进行事件/日志收集。

为什么要对这些操作进行统计呢？因为这些数据能够说明某一个用户在一个特定的时间段对哪些商品感兴趣，预计对哪些（或哪一类）商品会产生购买订单。借助后端的数据分析手段，还能知晓某一类用户对哪一类商品感兴趣的配比概率。所以这些商品详情查看的用户操作行为日志特别有商业价值。

日均 200 万 PV 是一个什么概念呢？这么说吧，翼支付（bestpay.com.cn）的日均 PV 在 34 万左右，汽车之家（autohome.com.cn）的日均 PV 在 100 万左右，折 800（zhe800.com）的日均 PV 在 600 万，携程在线（ctrip.com）日均 PV1200 万，京东（jd.com）日均 PV3.7 亿，淘宝（taobao.com）日均 PV6.4 亿（以上数据均来自 2016 年 alexa.cn）……

PV 是 Page View 的简称，即一次页面的完整打开算作一次 PV。PV 的统计中，这次页面访问“是由哪个访问者发起的”并不会对统计结果构成直接影响，也就是说即使是同一个访问者连续两次打开同一个页面，也会算作 PV=2。这里要注意的另外一个问题是，由于在浏览器页面上会有很多访问链接（例如：多个图片链接、多个 AJAX 请求等），所以一次 PV 可能会包含多次对服务端的请求。

作为架构师，你的工作职责就是为这个日志记录系统设计一个易于业务扩展和技术扩展的软件架构。所谓易于业务扩展是指：也许在未来的某个日子不只是“商品详情模块”会集成本系统，用户中心模块也会集成本系统，又或者订单子系统也会集成本系统，你设计的“日志收集子系统”应该可以在未来被这些子系统轻松集成，而不需要修改“日志收集系统”的任何代码（目标子系统也只需要修改极少的代码，甚至不修改代码）。

所谓技术扩展主要是说“事件/日志收集”系统支撑的数据吞吐量可以进行可靠的横向扩展，而不需要停止服务或者要求业务系统进行改动，毕竟要相应考虑以上业务扩展中所描述的多种业务系统后续可能进行集成的情况。另外，由于未来很多第三方系统都需要进行集成，作为架构师的你不可能知晓这些第三方系统会使用的是什么编程语言，更不可能限制第三方系统必须使用哪些编程语言。所以在进行“日志收集系统”的设计时，需要考虑一种兼容各种编程语言的设计思路。

## 11.2 Kafka 应用场景一：侵入式方案

我们先来看看此问题的第一种解决方案。如果确定将要集成“事件/日志收集”系统的所有第三方业务系统都有良好的代码结构（当然实际工作这种情况不太可能），那么为这些第三方系统提供相应编程语言的客户端 API，就是一个可选择的方案，如图 11-2 所示。

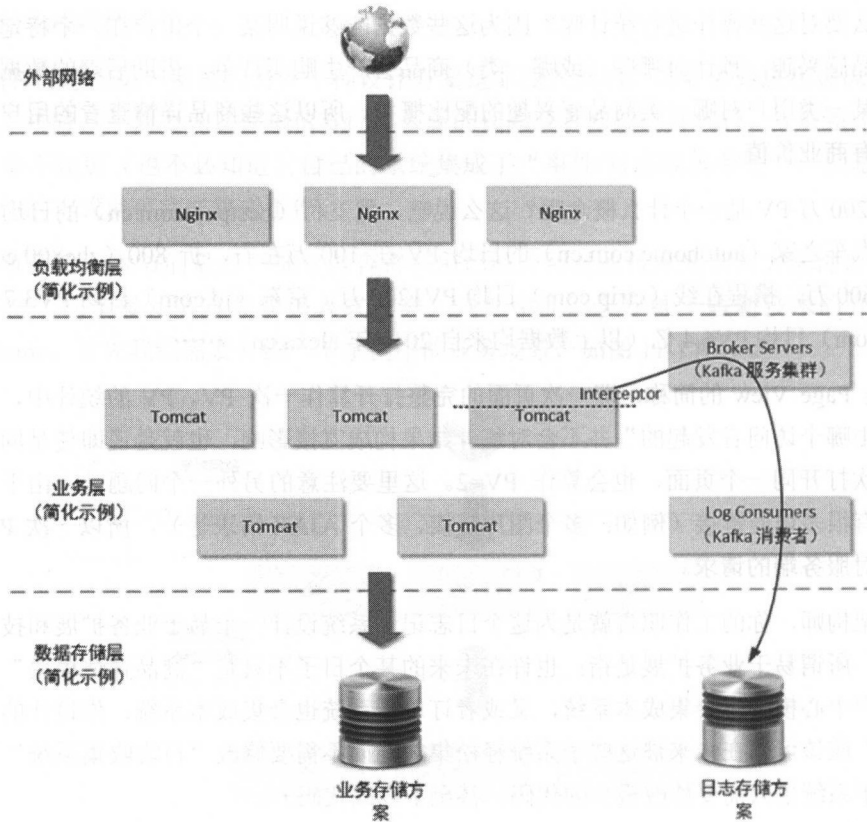


图 11-2 侵入式方案

在业务系统上使用过滤器/拦截器的方式对需要进行收集的访问请求进行拦截，分离出访问地址、访问用户、访问时间等重要信息后，将其作为 Kafka 消息发送给 Kafka Broker 集群。这些信息将最终到达由若干 Kafka Consumer 节点组成的处理服务，并使用适当的存储方案直接存储到连续文件中（存储到 HBase、Cassandra 这样的数据库中也可行，具体看这些日志数据将会被用于怎样的分析场景）。

11.2.1 设计重点

图 11-2 中，主要的展示目的是“事件/日志收集”系统在业务系统端是怎样被集成的。所以关于“事件/日志收集”系统的结构就画得比较简单。只给出了两个区块“Broker Server”和“Log Consumer”，下面我们重点分析一下本方案中的“事件/日志收集”系统的核心结构，如图 11-3 所示。

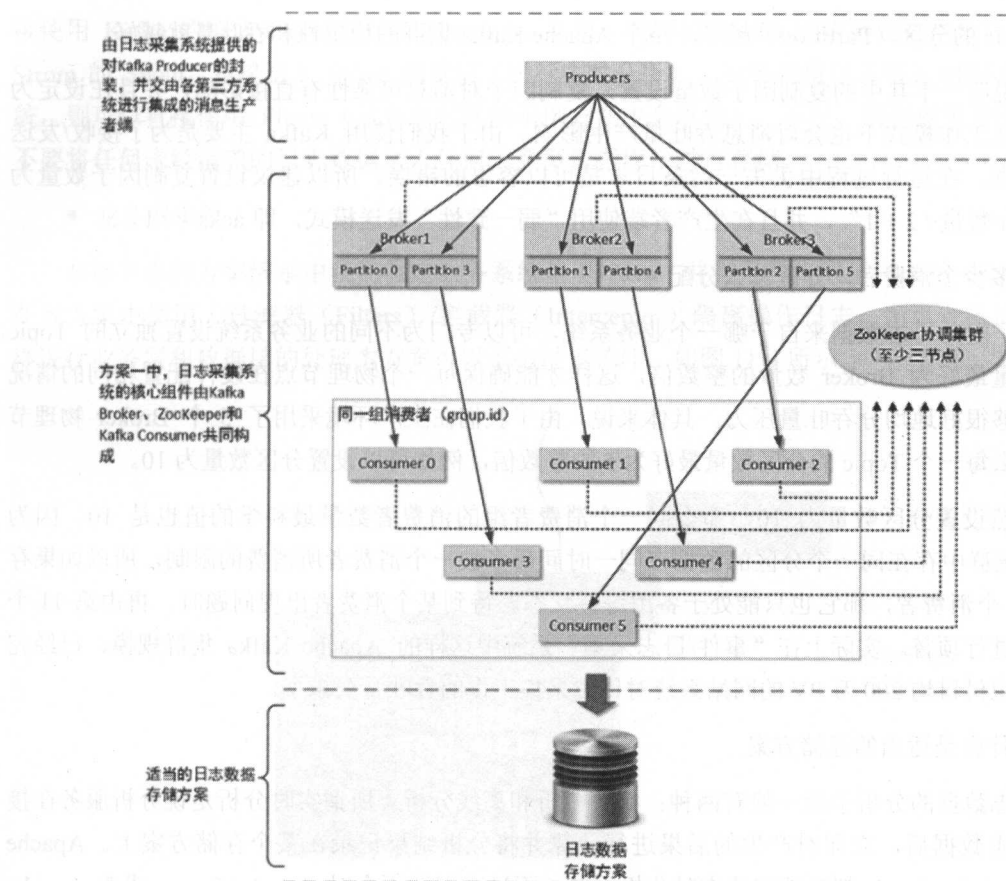


图 11-3 “Broker Server” 和 “Log Consumer”

- 需要多个 ZooKeeper 节点

使用 Apache Kafka 时，如果只使用一个 ZooKeeper 服务节点，那么整个集群也能正常工作。但是由于单个节点的 ZooKeeper 服务基本上没有容错能力，一旦单个 ZooKeeper 节点由于各种原因宕机，整个 Apache Kafka 集群就会崩溃。所以建议在生产环境下，至少为 ZooKeeper 服务准备三个服务节点，这样当某个 ZooKeeper 服务节点出现故障时，整个 Apache Kafka 服务还可以正常运行（三个节点的 ZooKeeper 服务最多允许一个节点发生故障）。

- 需要多少个 Broker

在生产环境下为了保证整个 Kafka 集群的稳定，请至少使用 3 个 Broker 物理节点。考虑到后期多个业务系统可能会使用“事件/日志收集”系统，那么可以在首次设计时将 Broker 设定为 5 个 Broker 物理节点。在 9.6 节中我们已经详细介绍过 Apache Kafka 的工作原理，Broker 越

多、Topic 的分区 (Partition) 越多, 整个 Apache Kafka 集群的稳定性和吞吐量就越好。

再说明一下其中的复制因子数量设置, 复制因子对消息可靠性有直接影响, 并且在设定为强一致性工作模式下也会对消息吞吐量产生影响。由于我们使用 Kafka 主要是为了接收/发送日志数据, 在运行过程中丢失一两条日志是可以容忍的错误。所以建议设置复制因子数量为 “Broker 数量 / 2 + 1”, 并且在生产者端使用 “弱一致性” 发送模式, 即  $acks = 1$ 。

- 多少个消费者, 分区怎么分配

为了区分日志数据来自于哪一个业务系统, 可以专门为不同的业务系统设置独立的 Topic。分区数量最好为 Broker 数量的整数倍, 这样才能确保每一个物理节点在硬件配置相同的情况下, 能够很好地均分吞吐量压力。具体来说, 由于我们在生产环境采用了 5 个 Broker 物理节点, 那么每一个 Topic 的分区数量最好为 5 的整数倍, 例如可以设置分区数量为 10。

既然设置分区数量为 10, 那么同一个消费者组的消费者数量最科学的值也是 10。因为 Kafka 集群中存在同一个分区的数据在同一时间最多被一个消费者所消费的限制, 所以如果存在第 11 个消费者, 那它也只能处于备用等待状态。待到某个消费者出现问题时, 再由第 11 个消费者进行顶替。实际上在 “事件/日志采集” 系统中这样的 Apache Kafka 集群规模, 完全可以应付日均 200 万 PV 的网站系统对日志采集工作的吞吐量要求了。

- 什么是适当的存储方案

日志数据的分析手段一般有两种: 实时分析和离线分析。所谓实时分析是说分析服务在接收到日志数据后, 立即对产生的后果进行计算并将分析结果记录在某个存储方案上。Apache Storm/Apache Spark 都是常用的实时分析系统, 不过本书并不会对 Apache Storm 或者 Apache Spark 进行详细介绍, 毕竟这属于另一个知识领域了 (在笔者的个人博客中后续会有专题讲解数据分析领域的知识)。实时分析在生产环境中有很多应用, 例如根据用户的上线/下线日志对用户的在线数量进行实时统计; 根据商品的点击情况, 对商品的查看数量进行实时统计; 根据用户的页面跳转情况实时形成用户浏览轨迹地图。

日志数据的另一种分析手段是离线分析。即分析服务在接收到原始日志数据后并不做任何处理, 只是将原始数据按照预定的格式 (又或者就是数据本来的格式) 存储到某个位置。当某个时间周期到来或者具体的事件被触发时, 再由其他软件对这些数据进行分析。Apache Hadoop/Cloudera Hadoop 就是常用的离线分析工具。可以通过某种手段, 将原始的日志信息存储在 HDFS 文件系统上, 以便 Hadoop 进行离线分析。离线分析在实际生产环境中也有很多应用, 例如按照用户的商品浏览情况分析用户的购买趋势、利用商品关键词进一步分析适合销售的用户群体、利用商品库存和价格走势预测最佳补货时机。

无论是实时分析还是离线分析 Kafka 的下层系统 (组件) 都需要做存储操作。例如可以直



接使用 Kafka 的消费者将消息写入 Cassandra 集群，可以将 Kafka 接收到的数据作为 Apache Strom 的 Spout，直接送入 Strom 的管道（进行实时分析）。如果要将日志写入 HDFS 文件系统，则可以直接使用 Flume（这个在后续的示例方案中会讲到）。不过，请别做愚蠢的事情：不要将任何未经清理的日志数据送入任何关系型数据库中进行存储。

- 业务层实现

在接下来的方案演示中我们假定业务系统基于 Java，并且已经集成了 Spring 框架。由于在本方案中使用了过滤器（Filters）/拦截器（Interceptor）隔离操作日志，所以业务服务中怎样进行业务层和数据层的处理本方案可以不必过多关注，如图 11-4 所示。

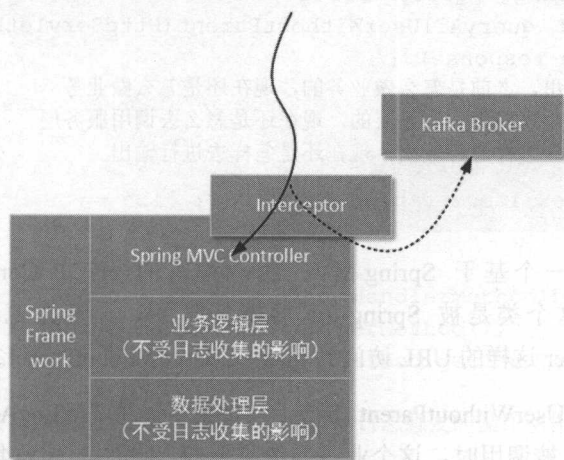


图 11-4 过滤器（Filters）/拦截器（Interceptor）的位置

这样做的好处是，可以将对日志的拦截操作在执行真正的业务操作前进行隔离，业务处理代码不需要关心在这之前都有多少层拦截，只需要按照原有的处理逻辑执行就行。

## 11.2.2 编码过程：生产者和业务系统集成

- 准备工作

演示的业务工程将使用 Spring-MVC 组件，所以如果需要查看演示效果，则请在工程中导入 Spring-MVC 组件（V3.2.x 的版本都行）：

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>3.2.10.RELEASE</version>
</dependency>
  
```

- 业务系统端集成

为了让更多的读者理解整个过程，首先来看一下这个拦截器的使用方式：

```
package templateSSHProject.controller;
.....
// Spring MVC 组件搭建的 HTTP 控制层
@Controller
@RequestMapping("/")
public class UserController extends BaseController {
    // 查询所有用户信息，但是不包括关联信息
    @LogAnnotation
    @RequestMapping("/queryAllUser")
    public void queryAllUserWithoutParent(HttpServletRequest request ,
        HttpServletResponse response) {
        // 在这里，之前是怎么做业务的，现在还是怎么做业务
        // 以前是怎样调用服务层的，现在还是怎么去调用服务层
        // 以前该怎样进行输出，现在还是怎样去进行输出
    }
}
```

以上代码片段是一个基于 Spring-MVC 组件编写的 HTTP Controller 层的类，名叫 UserController（当然这个类是被 Spring-Ioc 容器托管了）。在浏览器上可以使用格式为 `http://ip:port/queryAllUser` 这样的 URL 访问到 `queryAllUserWithoutParent` 方法。

请注意在 `queryAllUserWithoutParent` 方法上，使用了一个“@LogAnnotation”自定义注解。这个注解表示：当方法被调用时，这个业务系统需要向“事件/日志收集系统”发送日志信息。

- LogAnnotation 注解的定义

“@LogAnnotation”注解的定义非常简单，毕竟它只是一个标识，并不是整个结构能够运行起来的核心动力。

```
package templateSSHProject.controller.kafkaproducer;
.....
// 拦截标识注解。使用这个注解的方法说明需要向“事件/日志服务”发送消息
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogAnnotation {
    // 可以根据自己的需要，在这个注解中加入各种属性
    public String message() default "";
}
```

- 使用 Spring-MVC 的 Interceptor 拦截器，对 HTTP 请求进行拦截

为了让以上的代码能够运行起来，需要使用基于 Spring-MVC 的 Interceptor 拦截器，对

HTTP 请求进行拦截。让它在正式到达（执行）`queryAllUserWithoutParent` 方法前，能够先被拦截器预先处理。我们需要定义一个拦截器，如下所示：

```
package templateSSHProject.controller.kafkaproducer;

.....
//日志拦截器。一定注意，拦截器是基于 Spring MVC 的
// 如果使用的是 Struts 组件，那么就应该使用 Struts 提供的拦截器
public class LogMethodInterceptor extends HandlerInterceptorAdapter {
    // 由“事件/日志”系统提供的客户端工具包
    // 并且使用 Spring 进行代理的消息生产者服务对象
    // 而且已经在 Spring 配置中使用了 singleton 进行标记，
    // 说明全系统只有一个生产者服务对象
    @Autowired
    private ProducerService producerService;
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        //如果条件成立，则说明拦截器无效：因为我们只处理“方法级别的拦截”
        if (handler == null || !(handler instanceof HandlerMethod)) {
            return true;
        }
        HandlerMethod handlerMethod = (HandlerMethod)handler;
        Method method = handlerMethod.getMethod();
        LogAnnotation logAnnotation = method.getAnnotation
(LogAnnotation.class);
        //如果条件成立，则说明不需要进行事件/日志触发的操作动作
        if(logAnnotation == null) {
            return true;
        }
        //否则就要判断了
        Class<?> declaringClass = method.getDeclaringClass();
        String declaringClassName = declaringClass.getName();
        String methodName = method.getName();
        // 这是发送的日志数据，包括类名、方法名、调用时间
        // 当然还可以从 request 对象中提取更多的业务数据
        String message = declaringClassName + ":" + methodName + "[" +
new Date().getTime() + "]";
        this.producerService.sendMessage(message);
        return true;
    }
}
```

所有的 Spring-MVC Interceptor 都要继承一个父类：`org.springframework.web.servlet.handler.HandlerInterceptorAdapter`，当然 Interceptor 也是被 Spring-Ioc 容器托管的。为了使用 Interceptor，

需要在配置文件中加入相应的信息，如果使用的是 Spring-Boot，则请忽略这段配置直接使用理解就行了：

```
<!-- 请配置 xml 的 ns，加入新的 ns: xmlns:mvc="http://www.springframework.org/schema/mvc"
还有新的 schemaLocation:
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
-->
<mvc:interceptors>
    <bean
class="templateSSHProject.controller.kafkaproducer.LogMethodInterceptor"></b
ean>
</mvc:interceptors>
```

在 HandlerInterceptorAdapter 父类中，可以按照自身的需要选择性地重写 preHandle 方法、postHandle 方法、afterCompletion 方法或者 afterConcurrentHandlingStarted 方法。从这些方法名称就可以明白这些方法所代表的含义。这里我们选择重载其中的 preHandle 预处理方法。

请注意 HandlerInterceptorAdapter 类中定义的对象 “private ProducerService producerService”。这个对象就是由“事件/日志收集”系统提供的 Java 客户端开发包中的主要服务类。第三方业务系统需要使用这个服务类和“事件/日志收集”系统进行通信。

- 客户端开发包中的 ProducerService 定义和实现

以下是生产者接口定义：

```
package test.interrupter.producer;
// 生产者服务
public interface ProducerService {
    // 初始化 Kafka 生产端的配置信息
    public void init();
    // 向 Kafka Brokers 发送消息
    public void sendeMessage(String message);
}
```

以下是生产者接口实现：

```
package test.interrupter.producer;
.....
// 生产者服务实现
public class ProducerServiceImpl implements ProducerService {
    // Kafka 的 Brokers 列表
    private String brokers;
    // acks 的值，只能有三种：-1、0 还有 1
```



```

private Integer required_acks = 0;
// 请求超时间，默认为 10001
private Long request_timeout = 10001;
// Kafka 主服务对象
private Producer<byte[], byte[]> producer;
//分区数量
private Integer partitionNumber;
public void init() {
    // 验证所有必要属性都已设置
    if(StringUtils.isEmpty(this.brokers)) {
        throw new RuntimeException("至少需要指定一个 broker 的位置");
    }
    if(this.required_acks != 0 && this.required_acks != 1
        && this.required_acks != -1) {
        throw new RuntimeException("错误的 required_acks 值!");
    }
    if(this.partitionNumber <= 0) {
        throw new RuntimeException("partitionNumber 至少需要有 1 个");
    }
    Properties props = new Properties();
    props.put("metadata.broker.list", this.brokers);
    props.put("producer.type", "sync");
    props.put("request.required.acks",
this.required_acks.toString());
    props.put("request.timeout.ms", this.request_timeout.toString());
    ProducerConfig config = new ProducerConfig(props);
    this.producer = new Producer<byte[], byte[]>(config);
}
public void sendMessage(String message) {
    // 创建和发送消息
    byte[] messageContext = message.getBytes();
    KeyedMessage<byte[], byte[]> keyedMessage=new KeyedMessage<byte[],
byte[]>("MessageTopic", messageContext , null , messageContext);
    this.producer.send(keyedMessage);
}
// 以下是 set/get 部分，为节省篇幅就不再赘述了
.....
}

```

#### • 使用 Spring 托管 Producer 服务

作为 Java 开发的业务系统，至少有两种方式使用 11.2.1 节中定义的生产者服务接口。一种是在业务系统中的过滤器/拦截器中“new”这个类，然后手动调用初始化方法，最后再调用 sendMessage 方法；还好，示例中的业务系统使用了 Spring 容器，所以可以使用第二种方法：



将生产者服务注入容器，然后直接在过滤器/拦截器中调用 `sendMessage` 方法。

需要在业务系统的配置栏目中加入新的 `bean` 定义（如果使用的是 `Spring-Boot`，那么同样请忽略）：

```
.....
<!--
在这个示例代码的 Spring bean 配置中，一定要使用 singleton
否则你会发现 init 方法不断在执行，整个系统也会产生多个 producerService 对象
-->
<bean id="producerService" class="test.interrupter.producer.ProducerServiceImpl"
init-method="init" scope="singleton">
    <property name="brokers">
        <value>${kafka.producer.brokers }</value>
    </property>
    <property name="partitionNumber">
        <value>${kafka.producer.partitionNumber }</value>
    </property>
    <property name="required_acks">
        <value>${kafka.producer.required_acks }</value>
    </property>
</bean>
.....
```

和 `Kafka Brokers` 通信的主要参数放置在一个 `properties` 文件中，方便部署时进行更改（`kafka.properties`）：

```
kafka.producer.brokers=192.168.61.138:9092,192.168.61.139:9092
kafka.producer.partitionNumber=10
kafka.producer.required_acks=1
```

以上就是业务系统需要使用消息生产服务所进行的更改，以及生产服务自身是如何定义的。可以看到，在这种半侵入式的集成方式下，确实需要为集成“事件/日志收集”系统做很多的配置、编码工作。好的一方面是这些工作并不会影响原有的业务系统处理过程。

### 11.2.3 是否使用 `Spring Integration-Kafka`

`Spring Integration` (<http://projects.spring.io/spring-integration/>) 是依赖 `Spring` 核心框架进行工作的一套扩展组件。通过这套组件开发人员可以方便地在应用工程上集成第三方中间件技术，例如使用 `Spring Integration-Redis` 集成对外部 `Redis` 服务的调用，使用 `Spring Integration-FTP` 集成对外部 `FTP` 服务的调用，使用 `Spring Integration-Kafka` 集成对外部 `Kafka` 服务的调用。

`Spring Integration` 非常轻量、易于测试、入门文档较全、几乎没有使用门槛，只要知道

Spring 框架的基本使用方式就行。使用 Spring Integration 来实现对外部中间件服务的调用，大多数情况下比“自己编写”的解决方式都要好。

虽然 Spring Integration 框架非常好用，也确实节省了相当的集成工作，减少了错误调用的风险。但可能要让各位读者失望了：因为开发人员不能确定，将要集成“事件/日志收集”系统的所有业务系统都是基于 Spring 框架进行构建。所以在这样的背景下，提供给基于 Java（或者其扩展语言：Groovy、Scala）业务系统使用生产者服务，不应该和 Spring 形成强依赖关系，以保证在没有使用 Spring 框架的 Java 业务系统上也实现生产者服务的集成。

在本小节中我们展示生产者端的示例代码，并没有经过优化。例如虽然通过 Spring 框架分离了业务代码和日志发送代码，但是一旦 HTTP 请求到来，这些代码还是会在同一个线程运行。那么如果出现由于远端的 Kafka 服务拥堵导致的生产者发送缓慢的情况，就会影响到业务服务中对业务请求的处理速度。

要解决这个问题，可以在业务系统中为生产者服务开辟专门的处理线程池。利用线程池的 BlockingQueue 队列存储待发送的日志消息，利用独立线程进行日志消息的发送。不过这个解决办法并不是最好的，只能算是一个办法。因为生产者最终还是会占用业务系统紧张的 JVM 资源，最终还是会自身的异常状况转嫁给业务系统。

#### 11.2.4 编码过程：消费者端

存在于“事件/日志收集”系统内部的 Kafka 消息消费者端的代码工作也是非常简单的。Kafka 消息消费者的工作只是用来接收这些日志数据并且使用“适当的存储方案”将这些消息存储起来（或者送入另一处理组件，例如 Strom）。

下面给出一段消息消费者端的代码：

- 一个 ConsumerThread 对象就代表一个消费者

```
package com.test.logservice;
.....
// 消息消费线程
public class ConsumerThread implements Runnable {
    private KafkaStream<byte[], byte[]> stream;
    .....
    @Override
    public void run() {
        ConsumerIterator<byte[], byte[]> iterator = this.stream.
iterator();
        // 这个消费者获取的数据在这里
        // 注意进行异常的捕获
```

```

        // 如果有异常抛出但是又没有在方法中进行捕获,就会导致线程执行终止
        while(iterator.hasNext()) {
            MessageAndMetadata<byte[], byte[]> message = iterator.next();
            int partition = message.partition();
            String topic = message.topic();
            String messageT = new String(message.message());
            ConsumerThread.LOGGER.info("接收到: " + messageT + " 来自于
topic: [" + topic + "] + 第partition[" + partition + "]");
            // 这里就需要选择一种“合适的存储方案”
        }
    }

    public void setStream(KafkaStream<byte[], byte[]> stream) {
        this.stream = stream;
    }
}

```

- **KafkaConsumerLauncher** 基于 Spring 框架连接 ZooKeeper 并且启动消息消费者

```

package com.test.logservice;

.....
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;
import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

// 这是 Kafka 的 Topic 消费者
public class KafkaConsumerLauncher implements ApplicationListener
<ContextRefreshedEvent> {
    // ZooKeeper 连接地址串
    private String zookeeper_connects;
    // ZooKeeper 连接超时时间
    private Long zookeeper_timeout;
    // 分区数量
    private Integer consumerNumber;
    // 消息消费者处理线程池
    // 每一个消费者都是线程池中的一个线程
    // 且线程池中线程数量就是分区数量
    private ThreadPoolExecutor consumerPool;
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        ApplicationContext ac = event.getApplicationContext();
    }
}

```



```

//这里的条件保证启动 ZooKeeper 的连接和消费者线程的启动是在 Spring 框架
//完成初始化以后
if(ac.getParent() == null) {
    this.startConsumerStream(ac);
}
}
// 开启消费者线程
public void startConsumerStream(ApplicationContext context) {
    // =====首先各种连接属性
    Properties props = new Properties();
    props.put("zookeeper.connect", this.zookeeper_connects);
    props.put("zookeeper.connection.timeout.ms",
this.zookeeper_timeout.toString());
    props.put("group.id", "consumerGroup");
    //=====
    ConsumerConfig consumerConfig = new ConsumerConfig(props);
    ConsumerConnector consumerConnector =
Consumer.createJavaConsumerConnector(consumerConfig);
    HashMap<String, Integer> map = new HashMap<String, Integer>();
    String topicName = "MessageTopic";
    map.put(topicName, this.consumerNumber);
    Map<String, List<KafkaStream<byte[], byte[]>>> topicMessageStreams
= consumerConnector.createMessageStreams(map);
    // 获取并启动消费线程，关键就在这里：一个消费线程可以消费一个 Topic 中的多个
    // partition
    // 但是一个 partition 只能分配到一个消费者线程
    List<KafkaStream<byte[], byte[]>> streamList =
topicMessageStreams.get(topicName);
    // 为每一个消费者创建一个处理线程。并放置到线程池中运行
    // 注意：并不需要监控这些消费线程的运行状态
    // 因为没有消息接收的时候，线程就自然会在"iterator.hasNext()"位置等待
    for(int index = 0 ; index < streamList.size() ; index++) {
        KafkaStream<byte[], byte[]> stream = streamList.get(index);
        ConsumerThread consumerThread = (ConsumerThread) context.
getBean("consumerThread");
        consumerThread.setStream(stream);
        this.consumerPool.submit(consumerThread);
    }
}
// 剩下的代码都是 set/get 结构，为了节约篇幅就省略了
.....
}

```

KafkaConsumerLauncher 中我们一共为名叫“MessageTopic”的 Topic 创建了 10 个消费者。

就像讲解 Kafka 特性时提到的那样：消费者数量不要小于 Topic 的分区数量，可以多出一些消费者数量作为备用。这样才能保证每一个分区都有一个对应的消费者进行消费。如果在集群中，设计了 5 个消费节点作为消费者，那么也可以为每一个消费者应用程序创建两个消费者，这样一共也有 10 个消费者了。

另外注意，在 `KafkaConsumerLauncher` 中我们使用了一个线程池对象 `consumerPool`，并且使用了 Spring 框架进行了注入；我们创建具体的消费者线程也是依托于 Spring 框架完成的，所以才会有“`context.getBean`”这样的语句。它们的 xml 配置情况如下：

```
<!-- 消费者启动器 -->
<bean id="kafkaConsumerLauncher"
class="com.test.logservice.KafkaConsumerLauncher" scope="singleton">
    <property name="consumerNumber" value="10"></property>
    <property name="consumerPool" ref="consumerPool"></property>
    <property name="zookeeper_connects" value="192.168.61.138:2181">
</property>
    <property name="zookeeper_timeout" value="10000"></property>
</bean>
<!--
以下是服务器节点专门为数据处理准备的处理线程
因为只有 10 个生产者，所以线程池的大小是固定的，也无须使用无限队列
-->
<bean id="consumerPool" scope="singleton" class="java.util.concurrent.
ThreadPoolExecutor">
    <constructor-arg value="10" type="int"></constructor-arg>
    <constructor-arg value="10" type="int"></constructor-arg>
    <constructor-arg value="10000" type="long"></constructor-arg>
    <constructor-arg value="MILLISECONDS"
type="java.util.concurrent.TimeUnit"></constructor-arg>
    <constructor-arg ref="threadCacheQueue"></constructor-arg>
</bean>
<bean id="threadCacheQueue" class="java.util.concurrent.
SynchronousQueue"></bean>
<!--
消费者启动线程
一定注意：prototype 属性值，它代表着每次 getBean 就创建一个新的 ConsumerThread 对象
-->
<bean id="consumerThread" class="com.test.logservice.ConsumerThread" scope=
"prototype"></bean>
```



## 11.3 Kafka 应用场景二：调整侵入式方案

### 11.3.1 方案一的问题所在

方案一并不是最好的半侵入式方案，却容易理解架构师的设计意图：至少做到业务级隔离。方案一最大的优点在于日志采集逻辑和业务处理逻辑彼此隔离，当业务逻辑发生变化时，并不会影响日志采集逻辑。但是我们能为方案一列举的问题却可以远远多于方案一的优点：

- 需要为不同开发语言分别提供客户端 API 包。11.2 节中我们介绍的示例使用 Java 语言，于是“事件/日志采集”系统就要提供 Java 语言的客户端 API 包。如果需要集成“事件/日志采集”系统的业务系统，都是公司内各个业务团队开发的，那么这个问题还算不上大问题——至少可以知道优先开发哪种语言的客户端，也知道需要开发哪几种有限的语言；但如果想将这个“事件/日志”采集系统发布成共享软件，或者上市进行售卖，那么这个问题将限制产品的快速发展。
- 由于“事件/日志采集”系统的客户端代码需要在业务系统中进行编码集成。所以 API 包的升级也是一个问题：重大的 API 包升级可能会造成之前版本的不兼容问题，导致业务系统重新更改采集系统的集成代码。同样，如果所有业务系统都在公司内部，那么这个问题也不大。但如果目标是要将系统产品化……
- 虽然在业务系统中，可以通过良好的代码结构将业务逻辑和日志采集逻辑进行隔离，但是日志采集的处理过程终归集成于业务系统中，或多或少会影响业务系统的处理过程。例如：当消息生产者速度减缓时，可能就会影响到业务系统的处理效率；当待发送的消息在业务系统端大量堆积时，这些消息就会大量占用本该由业务数据使用的系统内存。

### 11.3.2 方案二的解决思路

第二种解决方案中，我们只要求业务系统在页面上加载一段 JavaScript 代码，就可以完成业务系统的事件/日志采集工作。事件/日志数据通过 HTTP 协议，跨域传输到“事件/日志采集”系统（图 11-5）。

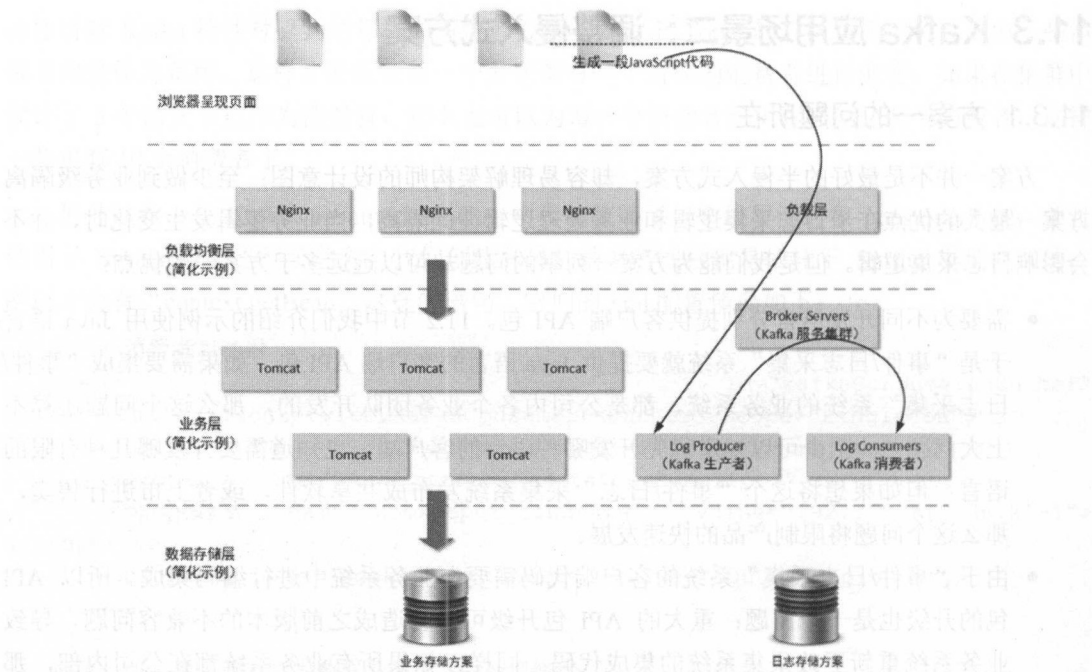


图 11-5 方案二整体思路

HTTP 协议的优势在于它是一个业内广泛使用的协议，下到刚从学校毕业的应届生上到有 20 年开发经验的资深工程师，都会使用这个协议。其次，这个协议与编程语言无关，你的业务系统无论是使用 JVM 虚拟机系列的语言进行开发的，还是使用 PHP 进行开发的，又或是使用 Node.JS 进行开发的，只要涉及在浏览器上呈现页面、发布基于 HTTP 的调用接口等技术操作，就会涉及 HTTP 协议。

在业务系统的页面集成 JavaScript 脚本实现对访问日志的采集的方式，实际上也有一定局限性：如果需要采集的事件不是针对页面访问进行的（例如采集业务服务器在设定的定时执行器中，进行了多少订单费用结算），那么这种方案二的方式就不太适用。还好，根据本章开始处描述的业务系统的日志采集需求，我们需要统计的恰好是商品订单的访问情况和商品价格走势的访问情况。

1. 负载层设计

方案一和方案二的负载层设计完全不一样。在方案一中，由于业务系统中集成了消息队列的生产者端，所以它的负载层完全由 Kafka Broker 中的分区（Partition）完成。但是在方案二中，由于业务系统向采集系统发送消息的方式是通过 HTTP 协议完成，所以采集系统的负载层需要进行相应的调整（图 11-6）。

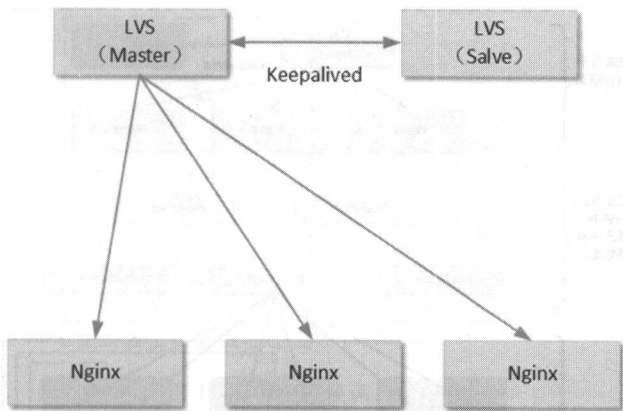


图 11-6 典型的互联网访问层设计

图 11-6 是一个典型的基于 HTTP 协议的负载均衡方案。在本书第 2 章和第 3 章中对这个方案有详细的介绍，这里就不再进行赘述了。如果觉得负载层太薄弱，还可以在其之上再加入 DNS 轮询等技术。

## 2. 为什么还要继续使用 MQ

第二种解决方案中，在“事件/日志采集”系统内部我们还是使用了 **Apache Kafka MQ** 技术，在采集系统内部进行消息的发送和接收。在一些读者看来，消息已经通过 HTTP 协议从外部业务系统（更确切来说是从业务系统用户的浏览器端）传输到了“事件/日志采集”系统内部，那么在采集系统内部只需要完成对这些原始日志的存储（或者送入及时分析系统）就行了，为什么还需要在采集系统内部采用消息队列机制呢？

考虑一下这种情况，当集成了采集系统的各个业务系统突然出现访问洪峰，产生大量的日志数据时。如果采集系统内部没有任何缓存机制，就会让采集系统成为整个顶层软件架构中的处理瓶颈。要知道，无论你在采集系统内部采用哪一种适当的持久化存储方案，都会消耗较多的处理时间。所以在方案二中，采集系统内部使用 **MQ** 队列就是出于缓存消息的目的。

当然也可以去掉 MQ，换成其他方案来缓存还不能处理的日志消息，但一定要有这样的缓存机制。因为处理单条日志数据，采集系统一般会消耗比业务系统多的时间，毕竟业务系统只负责发送日志数据。那么结合负载均衡层的调整和已有的 **Kafka** 消息队列的方案，我们就可以画出方案二中完整的系统架构图了（图 11-7）。

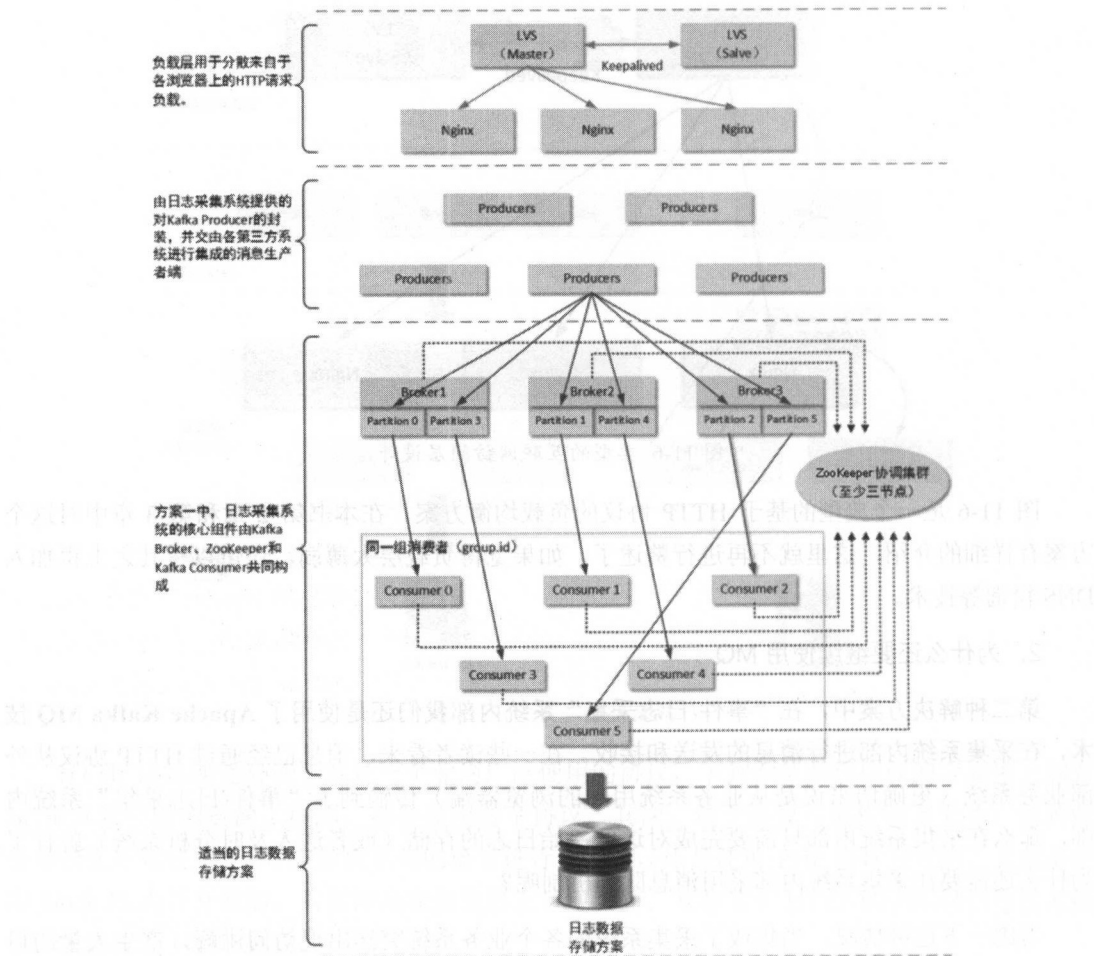


图 11-7 方案二技术方案细化

3. 跨域问题如何解决

在本方案中，业务系统通过呈现在浏览器上的页面，集成 JavaScript 脚本向采集系统发送 HTTP 请求。但是业务系统和采集系统很可能使用不同的域名（实际情况是作为“事件/日志采集”系统的架构师，你不可能控制业务系统的域名）。

如图 11-8 所示，跨域的情况下业务系统的页面不能通过浏览器端的 XMLHttpRequest 对象向工作在另外一个域的采集系统发送 HTTP 请求。为了解决这个问题，我们需要找到一种在浏览器端能够完成 HTTP 跨域调用的方法。好在靠谱的程序员们为我们提供了很多过往经验解决这个问题：proxy、Flash、iframe、Jsonp、CORS 等。这里我们根据采集系统的技术需求，介绍两种可以使用的解决办法：iframe 和 CORS。

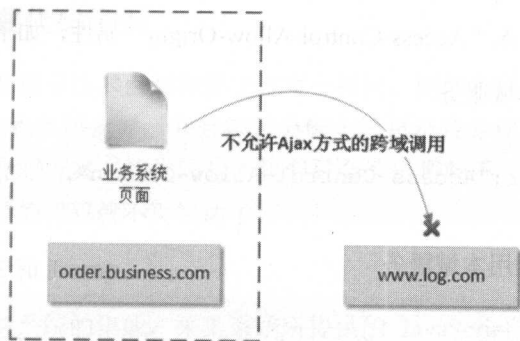


图 11-8 前端跨域调用的限制

### (1) CORS 方式

CORS 是 Cross-Origin Resource Sharing (跨源资源共享) 的简称。这个跨域技术主要由浏览器提供支持。当浏览器检查到 XMLHttpRequest 对象进行跨域调用时, CORS 会首先允许本次调用, 并且检查对方响应的 HTTP 协议的返回信息。如果返回信息的 Header 中存在 Access-Control-Allow-Origin 属性描述信息, 并且允许调用域, 那么就认为调用成功; 否则浏览器会提示类似于: “No ‘Access-Control-Allow-Origin’ header is present on the requested resource. Origin XXXXX is therefore not allowed access” 的错误。

由于 CORS 方式的跨域调用需要浏览器的支持, 所以存在一个浏览器版本的支持问题。如图 11-9 所示摘自 CORS 官网 (<http://enable-cors.org/>), 列举了各种浏览器版本对 CORS 的支持情况。

Cross-Origin Resource Sharing  LS													
Method of performing XMLHttpRequests across domains													
IE	Firefox	Chrome	Safari	Opera	iOS Safari	Android Browser	Opera Mobile	BlackBerry	Chrome for Android	Firefox for Android	IE Mobile	UC Browser for	
8	43	47	7.1	34	8	4.2-4.3	11.5	4.4	12				
9	44	48	8	35	8.1-8.4	4.4	12.1	4.4.3-4.4.4	12.1				
10	45	49	9	36	9.0-9.2	5.0-8.0	10	7	50	46	11	9.9	
11	46	50	9.1	37	9.3	5.0-8.0	10	7	50	46	11	9.9	
Edge	14	47	51	TP	38								
	48	52			39								
	49	53											

Resources:  
Mozilla Hacks blog post: Alternative Implementation by IE8 DOM access using CORS has.js test

图 11-9 CORS 的浏览器兼容情况

图 11-9 中黑底部分代表不支持 CORS 的浏览器版本, 浅色图块代表支持 CORS 的浏览器版本 (包括部分支持和完全支持)。要使用 CORS 的支持也很简单, 只需要在目标域的服务端



HTTP 协议 Header 部分写入 “Access-Control-Allow-Origin” 属性，如下 Java 代码所示：

- 允许任何域调用本域服务

```
.....
response.setHeader("Access-Control-Allow-Origin", "*");
.....
```

- 允许 XXXXX 域调用本域服务

```
.....
response.setHeader("Access-Control-Allow-Origin", "XXXXX");
.....
```

注意，如果使用 CORS 方式，并且服务前存在类似 Nginx 一样的 HTTP 代理服务，那么需要在 Nginx 的配置中增加对 Access-Control-Allow-Origin 的支持，类似如下 Nginx 配置片断：

```
http {
    .....
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Headers X-Requested-With;
    add_header Access-Control-Allow-Methods GET,POST,OPTIONS;
    .....
}
```

## (2) iframe 标签方式

使用 iframe 标签，实际上就是避免在浏览器端使用 XMLHttpRequest 对象。iframe 标签在各个版本的浏览器上基本上都没有不支持的问题，只有部分浏览器对 iframe 标签的属性支持有一些不同。以下是一个使用 iframe 标签调用另一域上服务的示例。

```
.....
<iframe style="display: none" src="http://192.168.1.100:9090/
templateSSHProject/ showSomething"></iframe>
.....
```

display 属性的作用是保证 iframe 标签不会有展示效果出现在最终页面上。使用 iframe 标签进行跨域调用是有明显缺点的：它会破坏前端开发人员既定的页面布局思路；如果不隐藏 iframe 标签，还会破坏开发人员在书写 JavaScript 脚本时的效果预判。

由于这两种方式都有一些问题，所以在实际操作中可以两种解决方法进行混用。首先判断当前浏览器版本信息，如果浏览器版本支持 CORS 方式，则优先采用这种方式（毕竟这种方式不会改变页面既有的 html 标签布局）；如果浏览器版本不支持 CORS 方式，则使用 iframe 标签方式。至于日志服务器所提供 HTTP 的调用接口上，始终都向 Header 增加 Access-Control-Allow-Origin 属性。

### 11.3.3 方案二的主要代码示例

由于解决方案二中有很多技术点都和解决方案一相同，例如都使用了 Apache Kafka MQ，都会使用 Spring 进行工程代码支撑，并且都不会影响消息消费者使用“适当的存储方案”进行存储。所以在本小节介绍方案二的代码时，我们只会给出那些不一样的，能够体现方案二工作特点的代码，其他部分的代码就不再赘述了。

#### 1. 混合采用 CORS 和 iframe

为了便于第三方业务系统的集成，采集系统所提供的 JavaScript 代码段应该尽量简单，最好就只需要业务系统引用一个 JavaScript 文件就行了。如下代码所示：

```
// 业务系统在页面上通过以下形式引用采集系统提供的脚本文件
.....
<script type="text/javascript" src="http://www.logsservice.com/analysis.
js?34ab834ea98ee838ac76ed3986347546"></script>
.....
```

以上代码片段中“www.logsservice.com”就是采集系统所在的域名，analysis.js 就是提供给各个业务系统进行嵌入的 JS 文件，“34ab834ea98ee838ac76ed3986347546”是一段由采集系统的“注册管理平台”生成的第三方业务系统的校验串，只有校验串所绑定的域名和当前嵌入 JS 文件页面所在的域名相同时，采集系统才认为本次采集数据有效。以下为“analysis.js”文件的脚本代码示例（JavaScript 代码）：

```
var _supportchromeversion = ["47","48","49","50","51","52"];
// 首先，无论使用哪种方式向采集系统发送 HTTP 数据，都需要得到页面上引用本 JS 文件时传递
// 的校验串 encrypted
// 这个 encrypted 参数含有相当的信息量
// 日志服务通过这个 encrypted 验证用户权限，业务系统域名匹配等信息
var encrypted = null;
var scripts = document.getElementsByTagName("script");
for (var index = 0; index < scripts.length; index++) {
    var script = scripts[index];
    // 如果条件成立，则说明找到了在页面上 JS 文件的引用位置，并且有加密参数记录
    if (script.src.indexOf("js/analysis.js") >= 0 && script.src.indexOf
("?") >= 0) {
        encrypted = script.src.split('?')[1];
    }
}
// 如果没有传递 encrypted 信息，则认为是错误的 JS 引用。不再进行处理
if(encrypted != null && encrypted != "") {
    // 确定当前浏览器是否支持 CORS 方式
    var bowersInfos = getVersion();
    var supportCors = false;
```

```

// 在本示例中,我们只判断了 chrome 浏览器的版本信息
// 其他浏览器版本的判断原理相似
if(browsersInfos.browser == "chrome") {
    var currentVersionArray = browsersInfos.ver.split(".");
    var currentVersion = currentVersionArray[0];
    if(contains(_supportchromeversion , currentVersion)) {
        supportCors = true;
    }
}
// 这里可判断其他浏览器的支持情况
// 如果支持,则直接使用 XMLHttpRequest 发起请求
// 时间戳是为了防止 HTTP 304
var timestamp = new Date().getTime();
if(supportCors) {
    var req = createXmlHttpRequest();
    var url = "http://127.0.0.1:9090/templateSSHProject/analysisSomething?
encrypted=" + encrypted + "&" + timestamp;
    req.open("GET" , url , true);
    req.send(null);
}
//如果不支持,则使用 iframe 方式进行请求
else {
    var context = "<iframe style=\"display: none\" src=\"http://
127.0.0.1:9090/templateSSHProject/analysisSomething?encrypted=" + encrypted
+ "&" + timestamp + "\"></iframe>";
    document.write(context);
}
}
// 获取浏览器版本的方法
// 该方法常用于测试使用。包括的浏览器并不完整
function getVersion() {
    var Sys = {};
    var ua = navigator.userAgent.toLowerCase();
    var re = /(msie|firefox|chrome|opera|version).*?([\d.]+)/;
    var m = ua.match(re);
    Sys.browser = m[1].replace(/version/, "safari");
    Sys.ver = m[2];
    return Sys;
}
//获取 XmlHttpRequest 对象
function createXmlHttpRequest() {
    if(window.ActiveXObject) {
        return new ActiveXObject("Microsoft.XMLHTTP");
    } else if(window.XMLHttpRequest) {

```

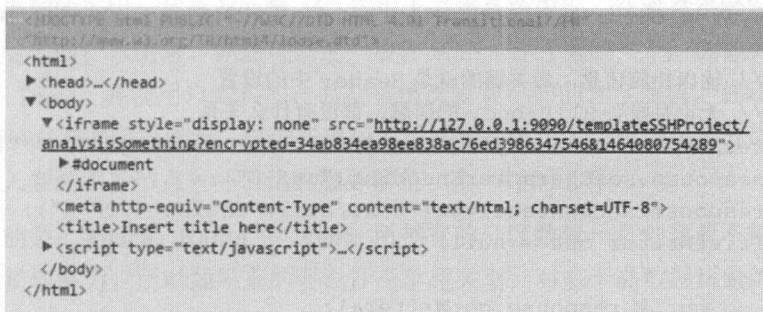


```

        return new XMLHttpRequest();
    }
}
// 用于集合元素比较
function contains(collection, obj) {
    var index = collection.length;
    while (index--) {
        if (collection[index] === obj) {
            return true;
        }
    }
    return false;
}

```

根据以上代码片段，如果浏览器不支持 CORS 方式，那么脚本代码将在页面输出一个 iframe 标签，并通过这个 iframe 标签完成跨域调用（当然这个标签在页面上是不可见的）。生成的 iframe 标签如图 11-10 所示：



```

<html>
  <head>...</head>
  <body>
    <iframe style="display: none" src="http://127.0.0.1:9090/templateSSHProject/analysisSomething?encrypted=34ab834ea98ee838ac76ed3986347546814640880754289">
      #document
    </iframe>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Insert title here</title>
    <script type="text/javascript">...</script>
  </body>
</html>

```

图 11-10 生成 iframe

如果浏览器支持 CORS 方式，那么脚本代码将创建 XMLHttpRequest 对象，并通过 XMLHttpRequest 对象完成跨域调用。注意：为了方便调试，以上实例代码中使用了一个笔者本地可调试的 URL，代替了“www.logsservice.com”。读者可以根据自己的 URL 进行替换。

## 2. 生产者编码

讨论了采集系统为业务系统提供的 JavaScript 脚本文件，我们再来说说采集系统的 HTTP 接口层代码：

```

package templateSSHProject.controller;
import java.io.PrintWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;

```

```

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import test.interrupter.producer.ProducerService;
// Spring MVC 组件搭建的 HTTP 控制层
@Controller
@RequestMapping("/")
public class AnalysisController {
    // 这里就是消息生产者对象
    // 其工作方式与方案一中的工作方式一致
    @Autowired
    private ProducerService producerService;
    // 做一些分析动作
    @RequestMapping("/analysisSomething")
    public void analysisSomething(HttpServletRequest request,
    HttpServletResponse response) {
        String param = request.getParameter("encrypted");
        // 利用 Kafka 生产者端发送消息
        this.producerService.sendMessage(param);
        System.out.println("public void sendMessage(String message) : "
+ param);
        // 输出相应信息，最关键的就是 Header 中的设置
        // 无论有没有 HTTP-body 的信息，都没有什么关系
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setCharacterEncoding("utf-8");
        response.setContentType("text/html; charset=UTF-8");
        PrintWriter out = null;
        try {
            out = response.getWriter();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        out.print("");
    }
}

```

采集系统保持高吞吐量的其中一个关键在于，Web 控制层中所使用的 Apache Kafka 消费者对象 `producerService` 能够快速地将消息发送出去。可以沿用方案一中对 Apache Kafka 消息生产者的设置。

### 11.3.4 其他设计思考

- 保证日志收集权限

按照解决方案二的设计思路，完成设计的“日志/事件采集”系统，是可以作为一款产品



对公众开放的。既然要开放系统就涉及各个用户的权限问题：至少应该保证用户 A 集成采集系统的业务系统是一个可用的业务系统，应该保证每一个用户只能在采集平台上控制自己的业务系统的统计信息。

采集系统可以提供业务系统注册功能，所有要使用采集系统的业务系统都首先需要通过注册页面进行注册。注册成功后，采集系统将会为这个业务系统生成一个唯一校验码。在进行日志采集时，只有校验码对应的业务系统和业务系统所注册的域名完全一致，采集系统才会认为本次数据有效。

- 卸掉流量洪峰

事件/日志采集系统架构设计的另一个重点问题，就是要保证“事件/日志采集”系统能够在多个业务系统同时出现流量洪峰的情况下，也能正常地进行日志统计，并且不影响各个业务系统的正常工作——不可能要求使用采集系统的各个业务日均 PV 不能超过××××的最大阈值。

除了上文提到的采用一款高吞吐量 MQ 作用于采集系统内部，在流量洪峰时堆积消息消费者还未来得及处理的日志消息。你还可以进一步在 Kafka 分区上做文章，例如为每一个业务系统创建独立的 Topic，并视用户购买的产品套餐情况设置不同的分区规模。你还需要为整个采集系统安排 40%左右的闲置资源，以便在出现流量洪峰的情况下，可以快速升级每个物理节点的性能或者加入新的服务节点——云化的服务器是一个不错的选择。

需要注意的是：Apache Kafka 中 Topic 所拥有的分区数量一旦创建就不能改变的缺点会限制它的横向扩容潜力。所以如果真的要设计一款超大型，对多个高数据流量的业务系统进行完全开放的采集系统，其中是否还是采用 Apache Kafka 作为核心消息传递手段就需要再进行慎重考虑了。

实际上在笔者专栏中，分布式系统中最关键的几个问题都已经有过介绍：服务节点发现方法、服务协调和选举规则、数据一致性的解决思路、网络 I/O 模型、缓存、异步处理。那么为什么不自己写一个满足技术需求的 MQ 呢？另外，阿里的开源项目 RocketMQ 也是一个不错的选择。

- 沿用方案一的 MQ 的设计

和解决方案一相比，在解决方案二中的消息消费者代码，包括其中调用的“合适的存储方案”都不需要做任何的变化。日志系统为业务系统提供的 HTTP 调用接口是为了保证各种业务系统的调用兼容性；继续在日志系统内部使用 MQ 是为了保证日志系统不会成为任何外部系统的调用瓶颈。这样，在解决方案二中就进一步优化了解决方案一中遗留的设计问题。

11.3.5 百度站长统计工具

类似方案二这样，在浏览页面嵌入 JavaScript 代码进行访问日志采集的典型应用之一就是百度推出的“百度站长统计工具”（<http://tongji.baidu.com/>）。要使用这个统计产品，首先需要注册一个用户信息，并且告知统计工具你需要统计的业务系统的工作域名。然后百度统计工具就会生成一段 JavaScript 代码，并且带有校验信息。如图 11-11 所示。

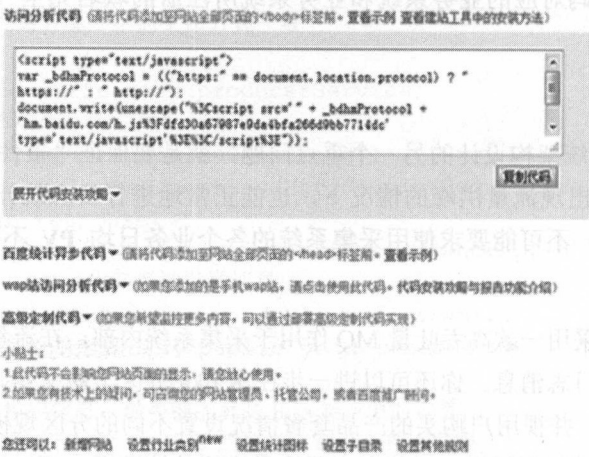


图 11-11 百度站长统计工具截图

实际上，如果你仔细阅读以上生成的代码，就会发现这段代码主要做的事情是：“通过这段代码生成另一个 JavaScript 引用标签”。最后只需要在你的业务系统页面上，加入这段 JavaScript 代码就行了（图 11-12）。



图 11-12 百度站长统计工具效果图

## 11.4 Kafka 应用场景三：非侵入式方案

以上两种方案中为了让业务系统能够集成日志采集功能，或多或少需要在业务系统端编写一些代码。虽然通过一些代码层次结构的设计，可以减少甚至完全隔离这些代码和业务代码的耦合度，但是毕竟需要业务开发团队花费精力对这些代码进行维护，业务系统部署时业务对这些代码的配置信息做相应的调整。

这里再为读者介绍一种非侵入式的日志采集方案。我们知道业务系统被访问时，都会产生一些访问痕迹。同样以“浏览商品详情”这个场景为例，当访问者打开一个“商品详情”页面时（URL 记为 A），Nginx 的请求接入日志（access 日志）就会首先记录 80 端口的访问日志，如果“商品详情”的信息并非全静态的，那么接下来业务服务上工作的业务代码还会在 Log4j 文件上输出相应的访问信息（如果开发人员使用了 Log4j 的话）。我们要做的事情就是找一款软件，将这些日志信息收集起来并存放在合适的位置，以便数据分析平台随后利用这些数据进行分析工作。

当然为了保证这些日志信息中有完整的原始属性，业务系统的开发人员和运维人员应该事先协调一种双方都认可的日志描述格式，以及日志文件的存储位置和存储规则等信息。

### 11.4.1 Apache Flume 介绍

Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application.

以上文字引用来自 Apache Flume 官网（<http://flume.apache.org/>）。大意是：Flume 是一个分布式的、具有高可靠的、高可用性的用于有效地收集、汇总日志数据的服务。它的架构基于数据流，简单灵活。我们要介绍的非侵入日志采集方案，就基于 Apache Flume 进行实现。

Apache Flume 非常简单，并且官方给出的用户手册已经足够你了解它的使用方式和工作原理（<http://flume.apache.org/FlumeUserGuide.html>），所以本书并不会专门介绍 Flume 的安装和基本使用，而是直接将 Flume 的使用融入到实例讲解中。如果你希望更深入学习 Flume 的设计实现，笔者还是建议你阅读 Flume 的源代码，在其官网的用户文档中已经给出了几个关键的实现类，通过这些实现类即可倒查 Flume 使用的各种设计模式，如图 11-13 所示。

Component Interface	Type Alias	Implementation Class
org.apache.flume.Channel	memory	org.apache.flume.channel.MemoryChannel
org.apache.flume.Channel	jdbc	org.apache.flume.channel.jdbc.JdbcChannel
org.apache.flume.Channel	file	org.apache.flume.channel.file.FileChannel
org.apache.flume.Channel	-	org.apache.flume.channel.PseudoTxnMemoryChannel
org.apache.flume.Channel	-	org.example.MyChannel
org.apache.flume.Source	avro	org.apache.flume.source.AvroSource
org.apache.flume.Source	netcat	org.apache.flume.source.NetcatSource
org.apache.flume.Source	seq	org.apache.flume.source.SequenceGeneratorSource
org.apache.flume.Source	exec	org.apache.flume.source.ExecSource
org.apache.flume.Source	syslogtcp	org.apache.flume.source.SyslogTcpSource
org.apache.flume.Source	multiport_syslogtcp	org.apache.flume.source.MultiportSyslogTcpSource
org.apache.flume.Source	syslogudp	org.apache.flume.source.SyslogUDPSource
org.apache.flume.Source	spooldir	org.apache.flume.source.SpoolDirectorySource
org.apache.flume.Source	http	org.apache.flume.source.http.HTTPSource
org.apache.flume.Source	thrift	org.apache.flume.source.thrift.ThriftSource
org.apache.flume.Source	jms	org.apache.flume.source.jms.JMSSource
org.apache.flume.Source	-	org.apache.flume.source.avroLegacy.AvroLegacySource
org.apache.flume.Source	-	org.apache.flume.source.thriftLegacy.ThriftLegacySource
org.apache.flume.Source	-	org.example.MySource
org.apache.flume.Sink	null	org.apache.flume.sink.NullSink
org.apache.flume.Sink	logger	org.apache.flume.sink.LoggerSink
org.apache.flume.Sink	avro	org.apache.flume.sink.AvroSink
org.apache.flume.Sink	hdfs	org.apache.flume.sink.hdfs.HDFSEventSink
org.apache.flume.Sink	hbase	org.apache.flume.sink.hbase.HBaseSink
org.apache.flume.Sink	asynchbase	org.apache.flume.sink.hbase.AsyncHBaseSink
org.apache.flume.Sink	elasticsearch	org.apache.flume.sink.elasticsearch.ElasticSearchSink
org.apache.flume.Sink	file_roll	org.apache.flume.sink.RollingFileSink
org.apache.flume.Sink	irc	org.apache.flume.sink.irc.IRCSink
org.apache.flume.Sink	thrift	org.apache.flume.sink.thriftSink
org.apache.flume.Sink	-	org.example.MySink
org.apache.flume.ChannelSelector	replicating	org.apache.flume.channel.ReplicatingChannelSelector
org.apache.flume.ChannelSelector	multiplexing	org.apache.flume.channel.MultiplexingChannelSelector
org.apache.flume.ChannelSelector	-	org.example.MyChannelSelector
org.apache.flume.SinkProcessor	default	org.apache.flume.sink.DefaultSinkProcessor
org.apache.flume.SinkProcessor	failover	org.apache.flume.sink.FailoverSinkProcessor
org.apache.flume.SinkProcessor	load_balance	org.apache.flume.sink.LoadBalancingSinkProcessor
org.apache.flume.SinkProcessor	-	

图 11-13 文档索引

11.4.2 设计方案

Flume 和业务服务系统在物理服务器上分别独立工作，在操作系统层面上是两个独立的进程，并没有任何关联。Flume 只对操作系统上的文件系统，或者指定的网络端口，又或者 RPC 服务进行数据流监控（Flume 中称之为 Source）。当指定的文件、指定的网络端口或者指定的 RPC 服务有新的数据产生时，Flume 就会按照预先的配置将这些数据传输到指定位置（Flume 中称之为 Sink）。这个指定位置可以是网络地址，可以是文件系统，还可以是另一个软件。Source 和 Sink 之间的数据流传输通告，称之为 Channel。

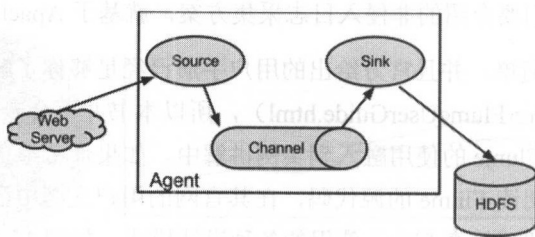


图 11-14 Flume 工作结构

图 11-14 来源于 Apache Flume 官方网站，是一个关于 Flume 中 Source、Sink 的例子。在这个例子中，Flume 采用一个 HTTP Source，用来接收外部传来的 HTTP 协议的数据；Flume 的 Sink 端采用 HDFS Sink，用来将从 Channel 中得到的数据写入 HDFS。那么基于 11.4.1 节介绍的 Apache Flume 工作特性，我们采用如图 11-15 所示思路进行日志采集方案三的设计。

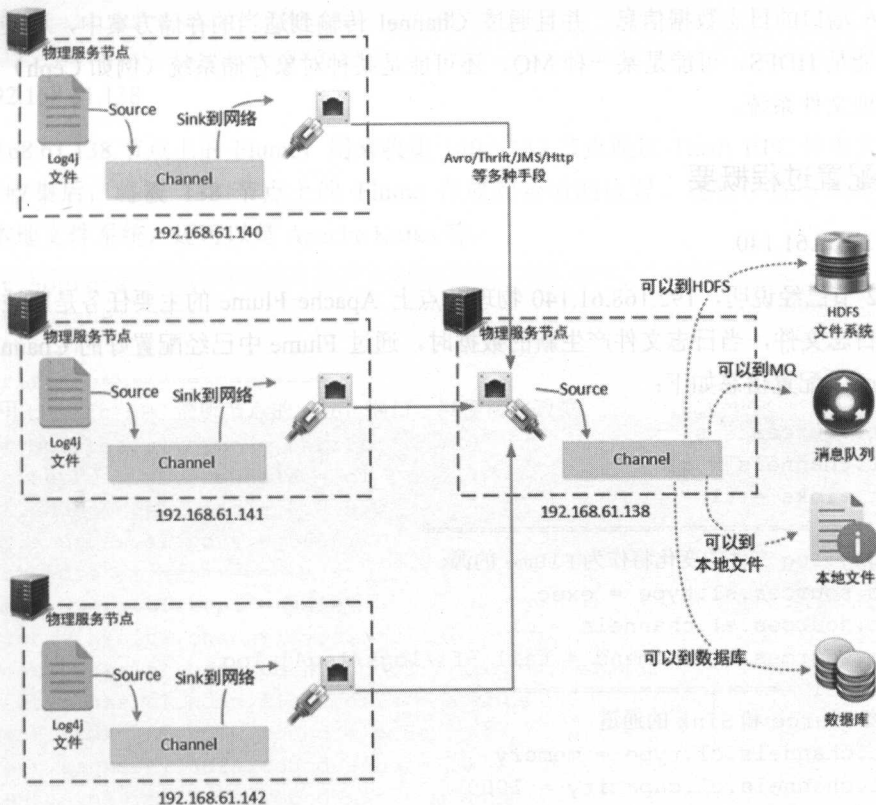


图 11-15 方案详细分解

### • 日志数据的来源和收集

图 11-15 中业务系统工作在 140、141、142 三个物理节点上，并产生 Log4j 文件。当然也可以直接使用 JBOSS、Tomcat 等服务的原生日志文件作为日志数据来源。有的情况下，我们需要对 Nginx 等代理服务上 HTTP 请求情况进行分析，那么可以使用 Nginx 的 access.log 文件作为日志数据的源是来源。还可以根据需要，在每一个物理节点上同时监控多个文件。

在 140、141、142 三个物理节点上，还分别安装了 Apache Flume。它们的工作任务都是一样的，即从指定的需要监控的日志文件中读取数据变化，通过配置好的 Channel 送到指定的



Sink 中。在图 11-15 的设置中是监控 Log4j 文件的变化，通过 Channel 使用 Thrift RPC 方式传输到远程服务器 192.168.61.138 的 6666 端口。

- 日志数据集中

物理节点 192.168.61.138 负责收集来自于 140、141、142 三个物理节点通过 Thrift RPC 传输到 6666 端口的日志数据信息。并且通过 Channel 传输到适当的存储方案中，这些适当的存储方案可能是 HDFS，可能是某一种 MQ，还可能是某种对象存储系统（例如 Ceph），甚至可能就是本地文件系统。

### 11.4.3 配置过程概要

- 192.168.61.140

11.4.2 节已经说明，192.168.61.140 物理节点上 Apache Flume 的主要任务是监控业务服务的 Log4j 日志文件，当日志文件产生新的数据时，通过 Flume 中已经配置好的 Channel 发送至指定的 Sink。配置信息如下：

```
agent.sources = s1
agent.channels = c1
agent.sinks = t1
# source =====
# log4j.log 文件的变化将作为 Flume 的源
agent.sources.s1.type = exec
agent.sources.s1.channels = c1
agent.sources.s1.command = tail -f /logs/log4j.log
# channel =====
# 连接 Source 和 Sink 的通道
agent.channels.c1.type = memory
agent.channels.c1.capacity = 1000
# sink t1 =====
# 通过通道送来的数据，将通过 thrift RPC 调用，送到 138 节点的 6666 端口
agent.sinks.t1.type = thrift
agent.sinks.t1.channel = c1
agent.sinks.t1.hostname = 192.168.61.138
agent.sinks.t1.port = 6666
```

192.168.61.141 和 192.168.61.142 两个物理节点也承载了业务服务，并且业务服务会将日志输出到同样的 Log4j 的位置。所以这两个节点上 Apache Flume 的配置和以上 140 物理节点中 Apache Flume 的配置一致。这里就不再对另外两个物理节点的配置进行赘述了。

另外需要注意的是 agent.sources.s1.command 配置的 Linux tail 命令。tail 命令可以显示当前文件的变化情况，如果带有 -f 参数，即表示从文件末尾的最后 10 行开始对文件的变化情况进行

行监控。如果这样配置，那么当 Flume 启动时，就会认为 Log4j 文件中已经存在的 10 行记录为新收到的日志数据，造成误发。要解决这个问题可以使用“-n”参数，并指定从文件的最末尾开始监控文件变化情况：

```
# 应该使用
tail -f -n 0 /logs/log4j.log
# 注意: tail -f /logs/log4j.log 命令相当于:
# tail -f -n 10 /logs/log4j.log
```

#### • 192.168.61.138

192.168.61.138 节点上的 Flume，用来收集 140~142 节点通过 Thrift RPC 传来的日志数据。这些数据收集后，将被 138 节点上的 Flume 存放到合适的位置。这些位置可以是 HDFS、Hbase、本地文件系统，还可以是 Apache Kafka 等。

```
agent.sources = s1
agent.channels = c1
agent.sinks = t1
# thrift =====
# 使用 thrift RPC 监听节点的 6666 端口，以便接收数据
agent.sources.s1.type = thrift
agent.sources.s1.channels = c1
agent.sources.s1.bind = 0.0.0.0
agent.sources.s1.port = 6666
# sink hdfs =====
# agent.sinks.t1.type = hdfs
# agent.sinks.t1.channel = c1
# agent.sinks.t1.hdfs.path = hdfs://ip:port/events/%y-%m-%d/%H%M/%S
# agent.sinks.t1.hdfs.filePrefix = events-
# agent.sinks.t1.hdfs.round = true
# agent.sinks.t1.hdfs.roundValue = 10
# agent.sinks.t1.hdfs.roundUnit = minute
# sink=====
# 为了检测整个配置是否正确，可先输出到控制台
agent.sinks.t1.type = logger
agent.sinks.t1.channel = c1
# channel=====
agent.channels.c1.type = memory
agent.channels.c1.capacity = 1000
```

以上配置文件中，为了查看这个采集系统的配置是否成功，我们将在 Flume 控制台作为 Sink 进行输出。注释的信息是 HDFS 作为 Sink 的配置。

#### 11.4.4 方案三细节说明

方案三中，最薄弱的位置是承担日志数据汇总任务的 138 节点。整个日志收集主架构中只存在一个这样的汇总节点，一旦 138 节点由于各种原因宕机，整个架构就将崩溃。即使 138 节点能够稳定工作，由于 138 节点同时承担多个物理节点传来的数据日志，那么它也极有可能成为性能瓶颈。所以我们需要找到一种缓解方案三中薄弱位置的办法。

##### 1. Flume 支持的高可用模式

还好，Apache Flume 为我们提供了非常简单实用的高可用模式：Load\_balance 模式和 Failover 模式。这两种工作模式都是对多个 Sink 如何配合工作进行描述。

- Load\_balance 模式

这种工作模式提供了多个 Sinks 负载均衡的能力。Load\_balance 会维护一个 active Sinks 列表，基于这个列表，使用 Round\_robin（轮询调度）或者 random（随机）的选择机制（默认为：round\_robin）向 Sinks 集中。基本上这两种选择方式已经够用了，如果你对调度选择有特别的要求，则可以通过继承 AbstractSinkSelector 类来实现自定义的选择机制。

- Failover 模式

这种工作模式提供了多个 Sinks 的故障转移能力。Failover 维护了两个 Sinks 列表，Failover list 和 Live list，在 Failover 模式下，Flume 会优先选择优先级最高的 sink 作为主要的发送目标。当这个 Sink 连续失败时 Flume 会把这个 Sink 移入 Failover list，并且设置一个冷冻时间。在这个冷冻时间之后，Flume 又会试图使用这个 Sink 发送数据，一旦发送成功，这个 Sink 会被重新移入 Live list。

为了保证能够为数据汇总节点分担性能压力，我们使用 Load\_balance 模式进一步演示对数据汇总节点的优化。

##### 2. 使用 Load\_balance 模式

从图 11-16 中可以看到在方案三的优化方法中，我们使用一个新的节点（192.168.61.139）和原有的 138 节点一起构成一组负载节点，共同承担日志数据的汇总任务。那么前端日志监控节点（140、141、142 三个节点）也需要做相应的配置文件修改。

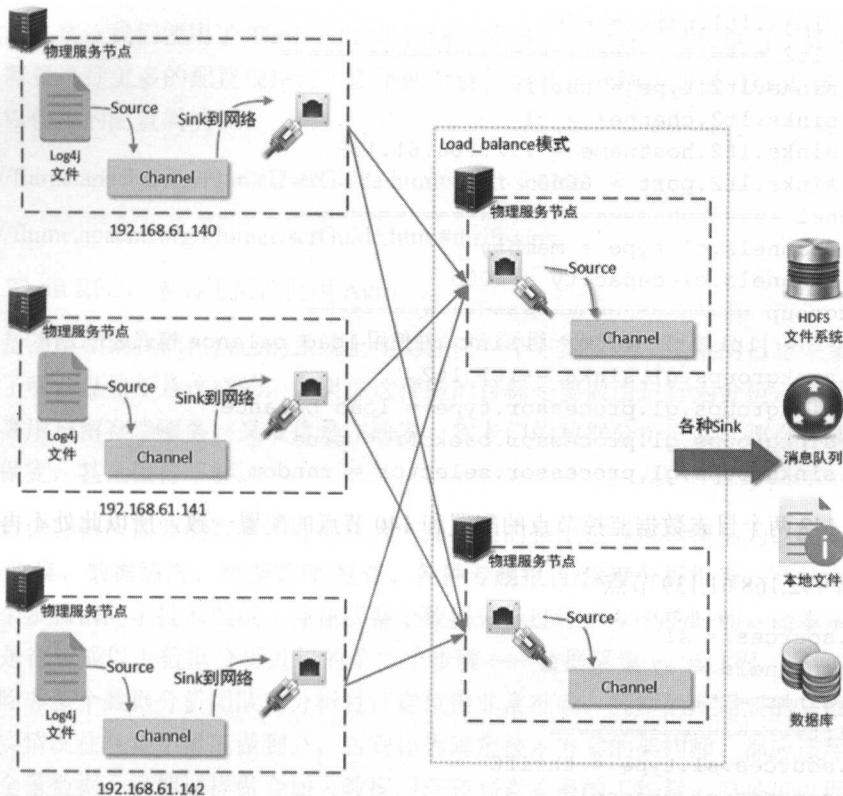


图 11-16 Load\_balance 模式

### 3. Load\_balance 配置过程

- 修改 192.168.61.140 节点

```
agent.sources = s1
agent.channels = c1
# 设置了两个 sink
agent.sinks = lt1 lt2
agent.sinkgroups = g1
# source =====
# 数据源还是来自于 log4j 日志文件的新增数据
agent.sources.s1.type = exec
agent.sources.s1.channels = c1
agent.sources.s1.command = tail -f -n 0 /log/log4j.log
# sink lt1 =====
agent.sinks.lt1.type = thrift
agent.sinks.lt1.channel = c1
agent.sinks.lt1.hostname = 192.168.61.138
```



```
agent.sinks.lt1.port = 6666
# sink lt2 =====
agent.sinks.lt2.type = thrift
agent.sinks.lt2.channel = c1
agent.sinks.lt2.hostname = 192.168.61.139
agent.sinks.lt2.port = 6666
# channel =====
agent.channels.c1.type = memory
agent.channels.c1.capacity = 1000
# sinkgroup =====
# 两个 sink: lt1 lt2 设置成一组 sink. 并使用 load_balance 模式进行工作
agent.sinkgroups.g1.sinks = lt1 lt2
agent.sinkgroups.g1.processor.type = load_balance
agent.sinkgroups.g1.processor.backoff = true
agent.sinkgroups.g1.processor.selector = random
```

141 和 142 两个日志数据监控节点的配置和 140 节点的配置一致, 所以此处不再赘述了。

#### • 新增 192.168.61.139 节点

```
agent.sources = s1
agent.channels = c1
agent.sinks = t1
# thrift=====
agent.sources.s1.type = thrift
agent.sources.s1.channels = c1
agent.sources.s1.bind = 0.0.0.0
agent.sources.s1.port = 6666
# sink=====
agent.sinks.t1.type = logger
agent.sinks.t1.channel = c1
# channel=====
agent.channels.c1.type = memory
agent.channels.c1.capacity = 1000
```

新增的 139 节点上 Flume 的配置信息和原有 138 节点上 Flume 的配置信息是一致的。这样保证了无论日志数据被发送到哪一个节点, 都能正确进行存储。

#### 4. 方案三的限制性

日志采集方案三也存在局限性: 这种方案不适合于开放性日志采集系统。也就是说, 如果你的日志采集系统需要像“百度站长统计工具”那样, 从设计之初的目标就是要发布给互联网上各个站点使用的, 那么这种基于操作系统日志变化, 并采用第三方软件完成采集过程的架构方案就不适用。



另外，方案三我们使用了 Thrift RPC 进行网络通信。这个方式是可以用于真正的生产环境的，但是需要进行更多的配置项指定。以下两个链接地址是分别是使用 Thrift 作为 Source 和 Sink 时可以使用的配置属性。

<http://flume.apache.org/FlumeUserGuide.html#thrift-source>

<http://flume.apache.org/FlumeUserGuide.html#thrift-sink>

除了 Thrift RPC，本书还推荐使用 Avro。

那么是不是如果我们在自己的系统上使用任何一个方案实现了全量的日志采集工作，就算完事大吉了呢？显然不是这样的，如果你这样做的目标是要做用户行为分析，并最终为改善产品体验提高用户留存率服务，又或你是在研发一款专门的数据分析产品，那么你要做的事情远远还没有做完，甚至做得不够。

收集数据分析所需要的数据，并最终为产品决策的整个过程可以概括为：**确定分析目标、数据采集/收集、数据清洗、数据管理/复查、各种专题报告/深度分析报告、产品洞察和决策。**而前面四个步骤偏向于技术领域，并花费整个数据分析过程中 90% 的时间。而本示例中的三个方案都只是在完成以上数据分析过程的第二个步骤——数据采集/收集过程。在数据采集/收集过程中，除非整个数据分析团队对分析目标定位得非常准确，列举的所需要的元数据类型非常清楚（现实情况往往是不可能做到），否则作为确定技术方案的架构师，都应该尽可能保证能够收集到全量数据。虽然这样做会加大数据清理和复查阶段的工作量，但是可以保证在数据分析师将数据带入公式时突然发现缺少了某些重要数据的情况下，不至于出现数据采集工作的大面积返工。

三个方案中采集的用户行为数据都不全面（以分析用户操作行为这个分析目标来看）：虽然三个方案都可以采集到客户端和服务器的请求交互情况，但这并不是所有的用户行为——因为有一部分用户行为是不会和服务器产生交互动作的。例如用户在客户端在终端程序上点击“home”键的情况，终端程序和服务端就没有交互，自然这部分行为就无法被采集。再例如，虽然本示例给出的三个方案都可以采集到某一个链接地址在某一时段被访问的次数，甚至是被不同用户访问的次数，但是并不能采集到用户是在哪些页面点击的这个链接。所以你需要在真实的业务环境下采集用户操作行为数据，那么必须需要前段团队进行配合才能保证达到相关分析目标而进行的数据采集的完整性。

## 第 12 章

# 场景实战：图片服务

图片服务系统是各种面向 C 端用户的业务系统常见的子系统，它的特点是存储规模大、请求频度高，且单张图片的读请求远远高于写请求。本章我们将从图片服务系统的需求分析开始，一起来讨论如何进行这类系统的技术选型、概要设计和详细设计，以及在这个过程中需要关注的技术重点。

虽然由于篇幅限制，图片服务系统中所涉及的分布式文件系统原理、非关系型数据库原理没有详细讲到，但这些知识点也并不是组成整个图片服务的所有关键点，也并不会影响本示例从设计思路层面对读者的启发。图片服务系统的讲述会分为四个步骤，前两节主要分析图片服务的需求、架构选型和技术方案，后两节内容进行性能优化和详细设计部分的讲解。由于不可能在书中给出全部代码，所以笔者会将整个示例工程的源代码放置在网络上，供有兴趣的读者进行下载：<http://download.csdn.net/detail/yinwenjie/9750136>。

### 12.1 需求场景

首先我们给出使用这个图片服务的场景，以便后续内容中根据这个需求场景进行初步架构设计、详细架构设计、技术选型，以及各个功能模块的设计等工作。这是一个中等规模的针对 C 端的电商服务站点，日平均 PV 量在百万级别（在国内属于中等规模的站点，例如从 Alexa 查询到大众点评的日均 PV 大致为 180 万、美团日均 PV 大致为 70 万、京东日均 PV 为 7500 万）。电子商务类型的站点，其特点是一张页面上会出现多张图片，而且基本上不会使用原图而是大量使用缩略图。另外，多个用户经常同时访问同一个页面，所以不但需要使用缓存，而且缓存一般都有多级，否则会产生大量重复的物理 I/O 请求。最后，这样的电商平台图片服务都属于顶级子系统，图片规模基本都是以 TB 计算，小一些规模的也有几个 TB，规模更大的

可能达到 PB 级别。

产品团队针对图片服务的基本功能要求是：需要支持单张或者多张图片上传，这样可以为站点运营人员/商户增强编辑体验，提高工作效率。另外需要在一些特定场景支持水印功能和图片特效功能，但这是优先级较低的功能，而图片的动态缩放则是必备功能，这是因为在站点的各种页面上都基本上不会使用原图而是使用各种等比缩小或中心截取的缩略图片，另外一个原因是一张图片会在各种客户端设备上显示，最常见的是 iOS、Android 和浏览器，而这些终端设备对图片的像素要求都是不一样的。产品团队针对图片服务的另一个基本要求是支持图片访问统计和报表。这个功能不是给终端用户使用的，而是给站点的运营团队准备的。通过访问统计和用户上传图片的报表统计，运营团队可以针对这些基础数据掌握每月的图片热点，而且这些数据可能还会为后期的数据分析所使用。

接着在与需求团队的沟通过程中，技术团队还推导了一些重要的非功能性需求，例如该图片服务属于整个站点的一个顶层子系统，所以需要保证 7×24 小时不间断运行，换句话说至少需要 99.999% 的稳定性；另外，图片服务的设计需要满足现有图片数据的割接，最好能够实现原有系统特别是已存放在磁盘上的图片的无缝割接；最后，整个系统至少需要 30% 的冗余存储空间，保证技术团队有足够的时间和空间进行后续的技术改造和扩容……

## 12.2 概要设计阶段

那么图片处理系统要关注的主要问题是什么呢？首先是高并发下的图片处理性能问题。图片处理是一项计算密集型操作，例如目前流行的图片缩放算法就有临近值法、双线性插值法、多项卷积法等，图片锐化算法多为拉普拉斯锐化法，图片增强算法可以采用中值滤波法、直方图均衡法或者幂变换法。这些算法都对 CPU 计算资源和内存资源有较高的要求，特别是高并发请求背景下。而 Java 语言相对于 C/C++ 语言在各种算法执行效率上又有先天的劣势，所以如果不能在性能上进行弥补就需要找到其他在高并发请求背景下加快图片处理效率的办法。其次是图片存储的问题，和存储稳定性安全性的问题。产生这个问题的原因已经在 12.1 节分析系统业务需求时进行了说明，这里就不再进行赘述了。接下来技术团队根据以上这些需求点和技术点，圈定了一个图片服务的大概的顶层架构思路，如图 12-1 所示。

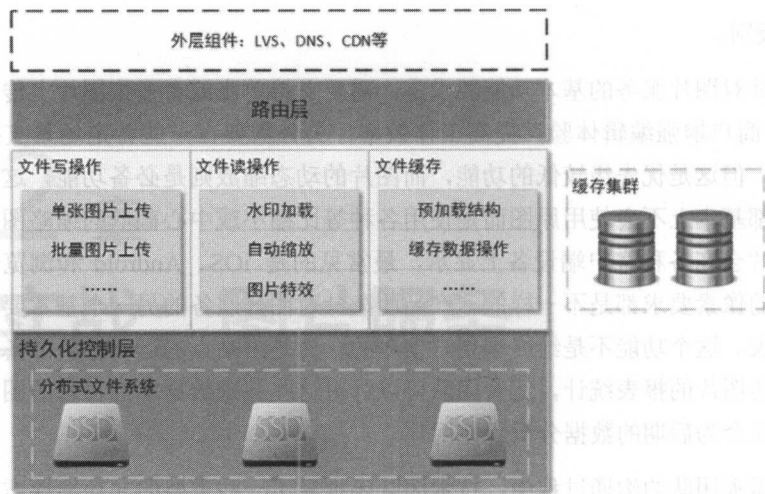


图 12-1 系统架构概要思路

虚线白底的外部组件表示那些可以和其他服务系统共同使用的组件，以及那些可以从第三方花钱购买的不需要技术团队关注的组件，图 12-1 中给出的这类组件包括：LVS、DNS/智能 DNS、CDN 等组件。接下来我们对图 12-1 中的各个服务层/服务模块进行选型分析。

12.2.1 分布式文件系统选型

分布式文件系统的选型是整个图片服务的重点，因为它提供了所有图片物理文件的存储基础。在分布式文件系统大量应用于生产环境前，解决此类问题一般使用网络块存储方案，例如 IBM Storwize V5000 单柜提供 24 个磁盘位，单柜支持最大 72TB 存储容量。但这个方案的缺点是单位容量价格非常高，另外虽然这个方案也可以较容易扩容，但是扩容费用也是极高的，基于单位建设成本的考虑，目前互联网公司都倾向于使用分布式文件系统甚至现成的云存储服务。如果采用分布式文件系统，那可以用的选型就太多了。这里我们先列举几种常用的分布式文件系统方案，然后针对图片文件的特性，再来对这些方案进行排除。这里可选择的分布式文件系统包括：HDFS、TFS、MFS、FastdFD 和 Ceph。

那么图片服务系统涉及的文件有些什么样的特点呢？

首先这些文件相对于视频文件、语音文件而言都不会太大，举个例子来说，虽然目前主流手机照相后产生的一张 JPEG 图片分辨率大致在 2500×3200 像素左右，大小在 2MB 到 4MB 之间，甚至更高端的手机照相后产生的 JPEG 图片文件大小可以达到 5MB 左右。但是考虑到网络流量和客户端速度体验等问题，上传到服务器的单张图片大小往往会控制在 2MB 以下（当然也有特殊的产品需求会要求上传完整的原始图片）。

另外这些图片的特点是读请求远远高于写请求，对大多数图片而言一旦上传到了服务器就不会发生更改操作了，甚至有的图片系统都不会提供对某张图片的修改功能。可是图片的读请求却会有很多而且带有一定周期性，什么意思呢？例如一张商品图片，当这个商品刚刚上架或者处于优惠期时，这张图片的读请求会处于一个高峰，而平时这张图片的读请求次数会处于一个平均水平，当这张图片对应的商品下架后其访问量就会很低了，但图片系统并不能想当然地删除这张图片，这是因为可能后端的运营团队后续进行商品管理和统计时，还会用到这张图片。

另一个特点是，虽然图片系统中单张图片的大小不会过大，但是整个图片系统的文件数量规模却会非常庞大。就像 12.1 节说到的：较小的文件规模也会有几个 TB，规模更大图片系统甚至可能达到 PB 级别。最后，图片系统都应该随时留有足够的冗余存储空间，以便应付至少几个月内系统的扩容要求。

通过以上的分析，大致可以得到图片系统中对于文件存储部分的要求：高稳定的工作特性，文件系统不能出现系统崩溃的情况——可以出现单节点故障，但是不能整个系统崩溃。这个要求决定了文件系统不能只有单个节点，而是多个节点协同工作：分布式文件系统是符合这样选型要求的。另外，文件系统的数据写性能可以不必太优秀，但是数据读性能必须要好，这样才能适应上面描述的读密集型业务。最后，文件系统的扩容应该比较方便，以便减少运维难度。在这个图片服务的示例中，我们将使用 Ceph 作为文件系统的选型。关于 Ceph 文件系统的介绍，可以参考其官网的介绍 <http://www.ceph.com/>。另外笔者的博客专栏上也有关于 Ceph 文件系统从安装、管理到原理的较详细介绍（<http://blog.csdn.net/yinwenjie/>）。

## 12.2.2 缓存系统选型

多级缓存方案在图片服务中是必然的存在，这是由图片服务的访问特性决定的。一般来说我们会设计三级至四级缓存，它们分别是客户端缓存、网络层缓存、路由层缓存和服务层缓存。如图 12-2 所示。

图 12-2 展示了一个四级缓存方案，其中基于 Nginx 的路由层缓存可有可无（后面会讨论原因），但是另外三级缓存在大多数高压力负载的图片服务中都是必须存在的，无非是采用哪种具体技术组件而已。除了之前提到的减少大量重复的物理 I/O 请求的目的，在图片服务中使用多级缓存方案还有以下这些原因。



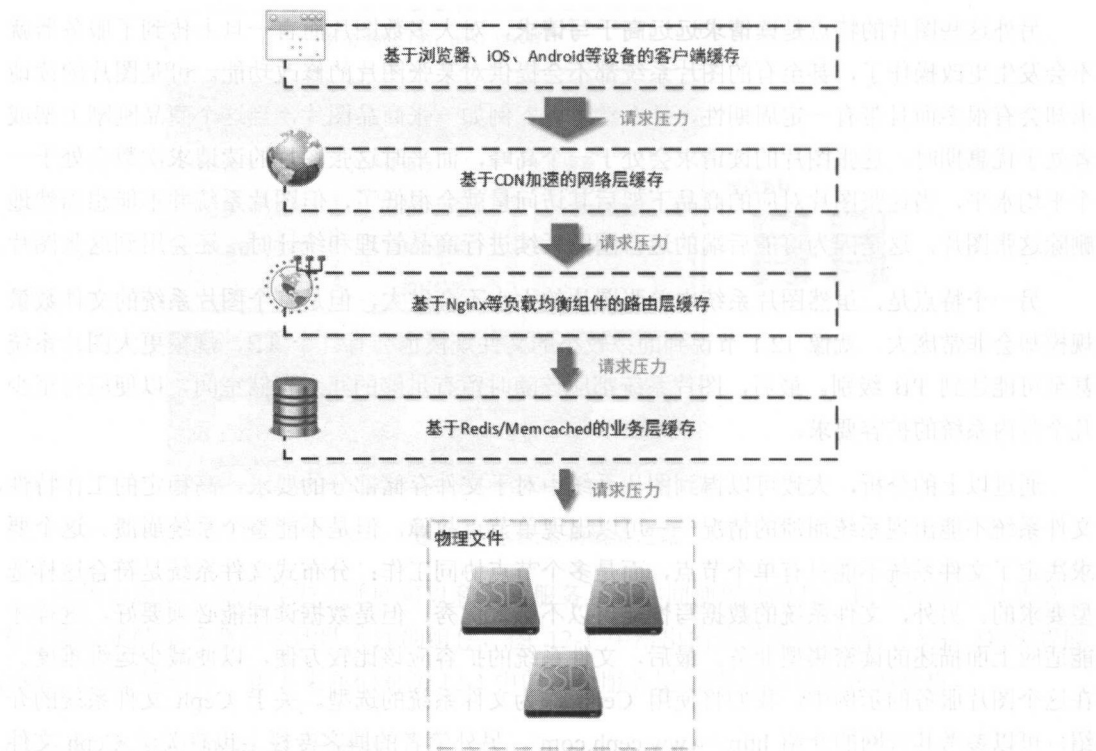


图 12-2 缓存分层

- 增加客户端访问速度：这是多级缓存方案中的客户端缓存和网络层缓存的首要目的，从访问效果来说客户端浏览器如果开启了缓存功能，当请求一张图片时，将从服务端上得到图片信息（CDN 相对于浏览器来说就算作服务器了），HTTP 的返回代码就是 200；当第二次请求同一张图片时，浏览器会自行判断在缓存区中的图片是否过期，也就是判断 HTTP 协议中的 `max-age/Expires` 属性，如果没有过期就直接到浏览器缓存中取得原始图片（如果发现过期就请求服务器，不过这时服务器发现资源没有改变，就会给客户端一个 HTTP 304 的代码，提示浏览器可以继续使用本地缓存）。CDN 的作用是解决服务端到客户端的最后一公里问题，当客户端向 CDN 请求图片时，CDN 会到“离客户端最近的”服务节点提取图片信息，如果那个服务节点上没有相应的图片，CDN 才会到真实的图片服务上提取图片并保存下来，以备后续使用。
- 拦截/缓解单个图片服务节点上的请求压力：在真实服务之前的路由层缓存，主要起到的就是这个作用。以 Nginx 上的 `proxy_cache` 为例，我们一般基于它使用内存 + 物理磁盘的方式缓存多种没有过期的静态文件资源，这其中就包括图片文件。
- 分散真实图片服务节点上的请求压力：例如路由层上的缓存功能，还起到分散请求压

力的负载均衡作用。无论技术团队在路由层使用的是 Nginx、Haproxy 还是 Spring Cloud——Zuul，要达到的一个目的就是下层多个图片服务节点做请求负载。由于路由层的定位问题，所以一些图片服务系统中即使路由层没有提供缓存功能，但也一定会提供负载均衡功能用来分散请求压力。

总的来说多级服务缓存最终以缩短客户端响应时间，减少图片服务器上真实的物理 I/O 操作压力为目的，经过多级缓存逐级削减向下传递的请求规模达到这样的目的。最终传递到最后一级业务层时，可能只剩下 10%~30% 的实际请求需要做真实的 I/O 操作（这还要看各级缓存的选型和详细参数设置）。而作为最后一级缓存的选型，除了读写速度上的要求，还有存储规模上的要求，所以 Redis 是一个比较理想的选择——利用 Redis 原生的 Cluster 技术，既可以兼顾访问速度、稳定性还可以获得很灵活的容量扩充方式。关于 Redis Cluster 的介绍可参考笔者的另一篇文章 <http://blog.csdn.net/yinwenjie/article/details/53905637>。

对缓存模块的设计和选型非常考验架构师对系统的驾驭能力。举个例子，为了保证数据 A 在靠近客户端的缓存模块中失效时，数据 A 的访问压力不会直接传导到最后端的 I/O 请求上，就要保证数据 A 在下一层缓存上在同一时间点上不会失效。基于这样的考虑架构师可能需要配合调整每一层的缓存过期时间，否则就可能出现数据 A 在最上层缓存失效的同时，所有层级的缓存都已经失效，数据 A 不得不直接在真实服务器的物理磁盘上重新读取，并重建每一级缓存的情况。但问题是，如果数据 A 的更新时间设置得过长，且每一级缓存的有效时长依次增大，那么在数据 A 真正发生变化后，这个变化可能需要很长时间才会体现在客户端上。所以并不是缓存层级越多越好，也不是缓存上的数据过期时间越长越好。

### 12.2.3 路由层选型

在这里笔者想结合一个在实际工作中遇到的问题进行讨论：目前 Spring Cloud 大行其道，很多新系统都开始基于 Spring Cloud 进行搭建，而且在实际工作中甚至有技术喊出了 Spring Cloud 中的 Zuul 组件完全可以取代 Nginx 的观点。那么真实情况是这样吗？

图片服务系统中，路由层有两个作用：缓存和负载均衡。这两个作用已经在以上的内容中给出了一个概要说明，这里就不再进行赘述了。基于对 12.2.2 节中需求的描述，技术团队主要考虑在两种技术组件中进行选择，它们是 Nginx 和 Spring Cloud——Zuul。本小节内容中我们主要对这两种技术对系统功能的契合度进行分析——它们在路由层分别使用时所基于的路由层功能定位完全不一样！Zuul 是 Spring Cloud 服务治理框架（也称为微服务治理框架）的一个重要组件，它可以单独使用也可以和 Spring Cloud 中的其他组件集成使用。表 12-1 给出了 Zuul 和 Nginx 对于图片服务需求各个方面的契合度。

表 12-1

组件	Nginx	Zuul	目前图片服务中的要求
缓存能力	Nginx 带有 Proxy Cache 模块，可以通过配置非常方便地进行数据缓存	需要自行实现	为了分散自上而下的数据请求压力，图片服务系统的代理层可能需要对数据进行缓存
反向代理能力	反向代理是 Nginx 的主要功能，配置灵活且性能优异	有反向代理功能，虽然配置灵活性没有 Nginx 好，但是通过编程可以很方便地进行扩展	没有特别要求
路由能力	支持基于正则表达式的路由功能配置	自带路由配置功能，支持通配符形式的路由配置。还可以通过 Zuul 中的 filters，扩展符合自身业务需求的路由规则	没有特别要求
负载均衡能力	自带非常强大的均衡功能，支持基于正则表达式的负载均衡配置，支持多种负载均衡规则	通过 listOfServers 关键字，可以配置负载均衡功能，但由于 Zuul 在 Spring Cloud 的定位问题负载均衡功能没有 Nginx 那么强大	需要较灵活的负载均衡配置能力，且对负载均衡功能的性能也有一定要求
流量控制能力	没有原生支持	没有现成支持，但通过 Zuul 的 filters 规则可以通过编程非常方便地实现	没有特别要求
安全控制能力	可以实现简单的安全控制功能，例如设置客户端黑白名单	安全控制是 Zuul 的主要职责之一，通过 Zuul 的 filters 规则可以通过编程非常方便地实现，另外还可以直接集成 Spring Cloud 的安全控制组件 Spring Cloud Security 来完成复杂的安全控制	没有非常复杂的要求，后期可能需要对图片盗链问题进行控制
监控和日志能力	有日志功能，包括访问日志、拒绝日志、异常日志在内的多种日志，可以选择开启也可以选择关闭	自带 Log4j 日志，通过结合 Flume 等数据汇聚组件可能非常方便地进行日志收集	目前没有特别要求，不过后续版本中可能会要求
编程扩展能力	支持技术人员使用 C/C++ 语言开发第三方 Module，并在 Nginx 编译安装时一并安装。但实际情况是，大家都只会使用一些现成的 Nginx Module	本来就是 Spring Cloud 微服务治理框架的一个组件，支持使用 JVM 系列语言进行开发，你可以通过 Zuul 为代理层加入任何想要的功能——如果不讨论软件解耦的科学性	没有特别要求

从表 12-1 我们可以看出，Zuul 和 Nginx 的功能虽然有一定的重合度，但是侧重点却是不一样的：Nginx 倾向于配置，Zuul 倾向于在特定规则下（filters 责任链）自行编程实现，究其原因两者的架构思路和在整体架构上存在的位置不一样。最终选择 Nginx 的原因实际上还是基于我们对路由层功能的定位：图片服务是单一的业务功能，并不存在再进行下层服务路由/代理的必要，所以没有必要使用 Zuul 提供的灵活路由规则支持。另外图片服务在至少能够预期的发展规划中并不存在很强的权限控制要求，即使有权限控制要求也是对图片盗链情况的控制，而图片盗链问题通过 Nginx 就可以很好地解决。最后，我们对路由层的功能定位主要就是缓存和负载均衡，而 Nginx 提供的负载均衡配置相对于 Zuul 提供的负载均衡更为灵活——这是因为 Spring Cloud 框架是一个服务治理框架，Zuul 可以直接把请求转向 Spring Cloud Eureka 服务注册中心，而且 Spring Cloud 框架内部的负载均衡可以依靠 Spring Cloud Ribbon 完成。

这里要特别讨论一下路由层的缓存控制问题。如果使用 Nginx，那么可以使用它自带的 Proxy Cache 功能建立的缓存空间；如果使用 Zuul 则需要自己通过代码实现一个缓存功能（图 12-3）。

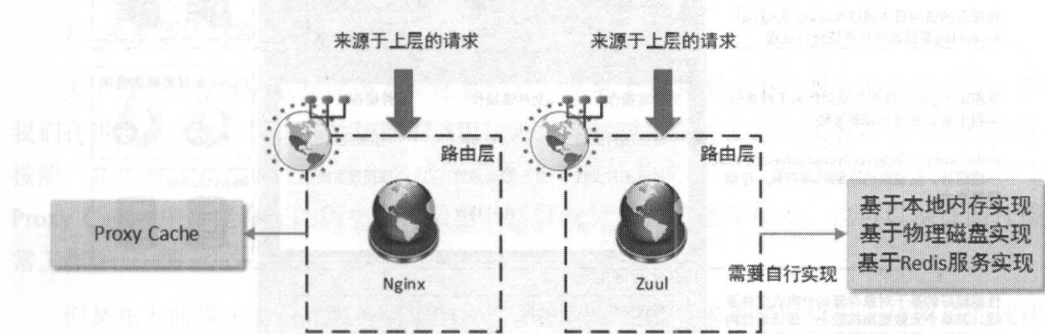


图 12-3 Nginx 和 Zuul

很明显，基于 Nginx 的方案更方便，因为是现成的；使用 Zuul 的方案需要自行开发缓存功能，所以会有额外开发工作量（而且不小，看实现到什么程度），却更能契合功能要求。那么实际情况是什么呢？实际情况是我们还需要考虑对缓存状态的控制力度。

使用 Nginx 的 Proxy Cache 虽然方便，但是它是独立工作的，只能按照配置好的方式载入载出数据资源。也就是说，当图片文件真正发生变化时我们无法通过 Nginx 提供的原生 API 接口，清除 Nginx 上相应的缓存数据。幸运的是，Nginx 提供了一个可选模块 `proxy_cache_purge`，通过 HTTP 请求的方式清空缓存，但这种主动清理方式在高并发情况下的性能并没有太多可靠性和性能方面的资料可查。这些原因还是其次，最重要的情况是我们无法在图片 A 变化时，准确知晓上层若干台设定了 Nginx\_Cache 的 Nginx 中哪些 Nginx 节点需要刷新缓存。所以最后

得出的路由层结论是：Nginx 存在的主要作用是负载均衡，可以为它配置 Proxy-Cache 模块，但是不能设置过长的有效时长，可以设置成 10 分钟但绝对不能设置成 10 小时。这样才能保证下层图片数据发生变化时，客户端被延迟通知的时间更短。

Nginx 和 Zuul 设计之初就针对两种不相同的业务领域，其功能定位和要解决的业务问题也是不一样的。之所以在路由层的技术选型中，比较这两种组件的差异，除了因为两者在功能上有一定的重合度，更重要的原因是：功能/业务要求决定技术选型，而不是反过来进行思考。

12.2.4 架构设计细化

基于以上所描述的各关键点的选型，最终我们可以将概要的架构设计进行细化。如图 12-4 所示。

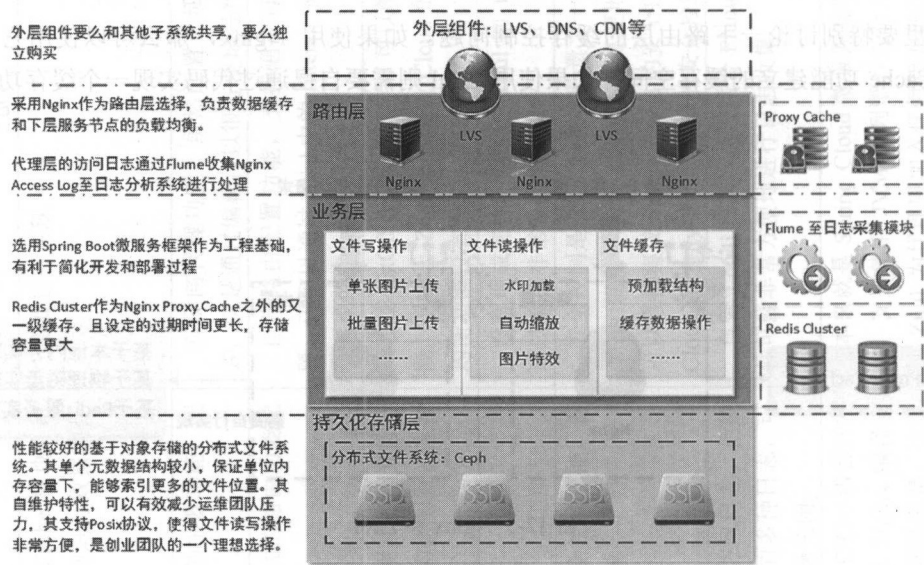


图 12-4 概要设计细化

图 12-4 中，最外层面向客户端访问加速的智能 DNS 路由、CDN 加速服务通过购买获得（目前市场上还有很多免费的服务可以使用），首先这些组件与图片服务的核心设计基本上没有什么关联，其首要目标是加快客户端的服务响应速度，另外这样做还可以有效减少运维团队的工作量。至于 LVS 组件，可以和其他独立工作的子系统共享使用。

我们最终采用 Nginx 作为图片服务的路由层，使用 Nginx 提供的负载均衡配置将数据请求压力分散到下层的多个服务节点上。使用 Nginx 原生的 Proxy Cache 技术作为客户端缓存（可选）、CDN 加速以外的第三级缓存，但不能将过期时间设置得太长（几分钟最合适）。如果



图片发生变更，也不能主动由业务节点基于 `proxy_cache_purge` 主动向 Nginx 通知删除缓存内容，因为根本不知道哪些 Nginx 节点需要被通知。类似图片文件访问频度这样的统计工作也在此路由层完成，只不过它不是由路由层本身来处理，而是使用类似 Flume 这样的日志收集组件对 Nginx 的 `access.log` 文件数据进行收集后，送至专门的日志分析系统完成。

为了提高业务层的开发过程，这次演示的开发工程将基于 Spring Boot 进行构建和代码编写，为了让图片文件的处理操作更加灵活，我们将基于责任链模式构建生产线形式的图片处理过程。我们还将业务层使用 Redis Cluster 构造最后一级缓存，这级缓存的过期时间是各层缓存中最长的，存储规模也是各层缓存中最大的。一旦图片文件被更新后，业务层服务将直接使用 Redis 的原生 Java API 删除缓存中对应的数据信息。

最后，持久层的分布式文件系统我们选用 Ceph。实际上这一层可用的选型是最多的，只要把握两个规则就行：相同的内存空间中可以存放更多的元数据，存储小文件时浪费的空间更少。你还可以选用 TFS、FastDFS，又或者直接使用块存储方案——光交换机 + 磁盘柜。

## 12.2.5 其他技术选型

### 1. 关系型数据库选型

关于持久化存储的数据库技术要注意一点，实际上它并不是图片服务的必要组件。例如，我们在进行设计时可以将图片访问的 URL 地址直接对应图片文件在服务器上的存储地址，并按照一定的规则将图片文件重命名成一个系统中唯一的文件名，最后再删除 Redis 和 Nginx Proxy Cache 中可能存在的历史文件数据。这样就算没有数据库技术，也可以保证图片服务正常工作。

但是在上面描述的图片服务需求中，产品团队还明确要求需要对用户上传图片的规律、活跃度等状态信息进行统计，需要对图片的物理磁盘读操作频度进行统计分析，所以一些结构化的数据还是需要持久化存储的。就拿图片的每日访问情况来说，当我们为通过 Flume 收集了 Nginx 的访问日志，并送入到一个独立的日志分析系统中进行处理后，类似单个用户每一天对图片数据的访问数量、每张图片在每一天的访问数量这样的分析结果还是要存入到数据库中以备后续的分析——无论是选用关系型数据库 MySQL、SQL Server 或者非关系型数据库 MongoDB、Apache Cassandra（图 12-5）。

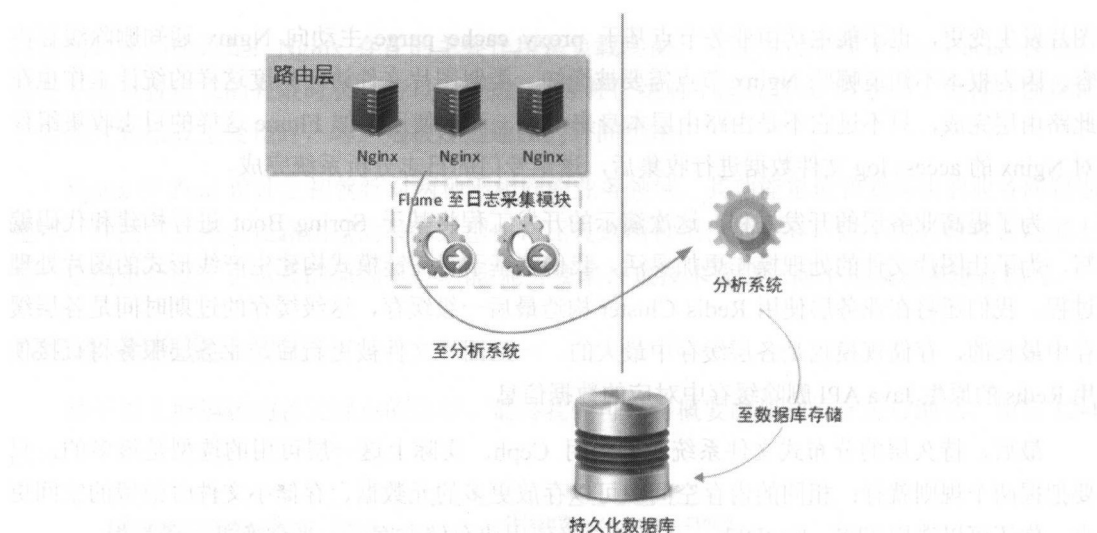


图 12-5 一种存储方案

也就是说，实际上要实现完整的图片服务系统的话还是离不开使用数据库技术的，但是这个基本上属于图片系统中的一个边缘技术组件，完全可以根据你所在公司某种默认规定的数据库技术作为依据。

## 2. 关于 Nginx 的补充

### • 关于 Nginx Proxy Cache

Nginx 的 Proxy Cache 缓存采用内存索引 + 物理磁盘存储的工作方式，所以为了进一步提高 Proxy Cache 的工作性能，在为 Proxy Cache 指定工作目录时，最好指定到一个独立挂载点上，并且这个挂载点的底层物理介质最好为 SSD 固态硬盘 + RIAD 5/RIAD 01 磁盘阵列。另外在之后我们介绍 Proxy Cache 配置时，还会讨论 Proxy Cache 的一些注意细节。

### • 关于 Image\_Filter 模块

有的读者可能会问，什么我们不直接基于 Nginx 提供的第三方模块 Image\_Filter 作为图片处理的基础呢？这个模块也可以实现图片的缩放、裁剪、翻转、特效等操作。是的，如果系统对图片特效处理的结果要求不高，完全可以使用 Image\_Filter 来动态处理图片请求。Image\_Filter 使用 C/C++ 语言完成，在完成单张图片同样的特效要求的前提下，处理性能也比使用 Java 原生的 Image I/O API 要高。但 Image\_Filter 可以提供的图片效果也是有限，例如 Image\_Filter 提供的特效方面只有透明度、锐化、旋转、变更图片质量等操作，但如果系统中有诸如效果增强、背景虚化等这样的图片特效要求，那还是只能由开发人员自行编程解决；Image\_Filter 虽然可以为图片加水印效果，但是要求水印图片背景必须透明。

## 12.3 关键技术点考量

### 12.3.1 责任链模式

在图片处理系统的首个版本中，我们计划先提供诸如图片等比例缩放、图片中心点裁剪、图片白化、图片文字水印等基本功能，但是为了保证软件设计部分能够在后续版本方便进行功能扩展，我们需要找到一种符合功能特点的行为模式，作为基本的设计模式。

我们在首期提供的这些功能并不能要求使用者（客户端）按照某种操作顺序执行，而应该由使用者自行确定操作顺序。什么意思呢？就是说不能规定使用者必须先缩放图片才能为图片添加水印，也不能规定要进行图片白化，就不能进行图片裁剪。而应该让使用者像使用 PS 软件一样——可以首先进行图片裁剪，然后再进行白化，最后再添加水印；也可以先向图片增加文字水印，然后再进行图片等比例缩放操作。我们可以用如图 12-6 所示的概要图表示这里文字描述的内容。

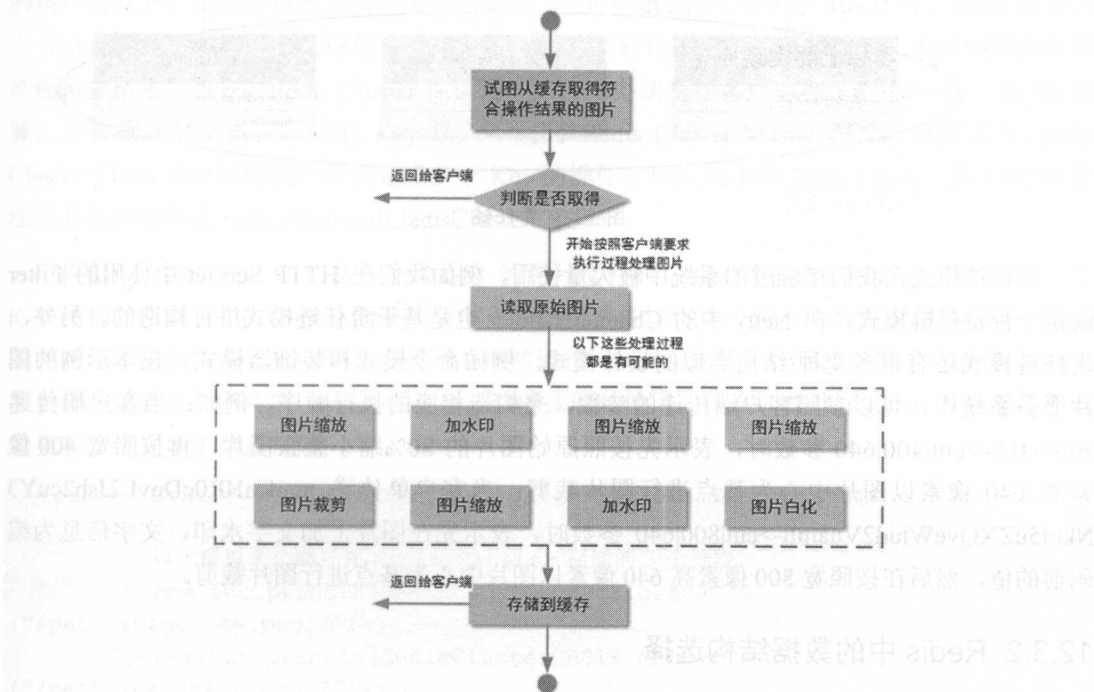


图 12-6 图片处理过程

从缓存中取得图片这一步的注意事项我们将在 12.3.2 节中进行介绍，这里我们先看读取原始图片后的处理过程。从图 12-6 我们可以看到读取完成后的图片各种处理组合，有一点类似

于生产线的概念，每一个图片处理器接收到上一个图片处理器的产品后，再按照自己的处理逻辑进行处理，最后输出到下一个处理器，这种表象性的处理特点符合一个典型的责任链模式所适应的处理场景。在责任链模式里，若干实际的处理过程被串成一个链式结构，数据在上一个处理器中被处理后传递到下一个处理器，每个处理器都按照自己的业务逻辑规则处理数据。如果责任链中某个处理器处理失败，则可以通过返回 `null` 或者抛出异常等方式通知整个责任链停止处理（图 12-7）。

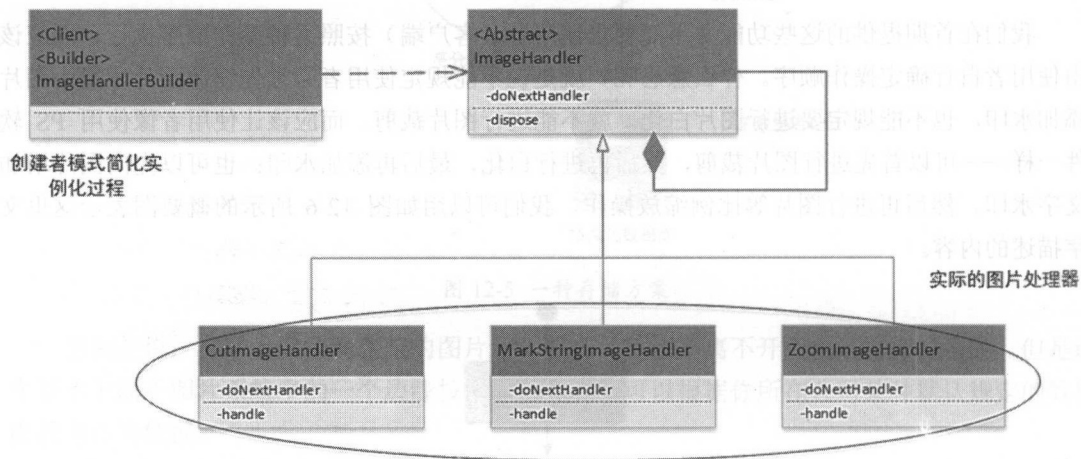


图 12-7 责任链

责任链模式在我们接触过的系统中被大量使用，例如我们在 HTTP Servlet 中使用的 Filter 就是一种责任链模式，在 Netty 中的 ChannelHandler 也是基于责任链模式进行构造的。另外，责任链模式还有很多变种/结构类似的设计模式，例如命令模式和装饰器模式。在本示例的图片服务系统中，可以参照客户端传递的参数，来构造相应的执行顺序。例如：当客户端传递 `zoom|0.8->cut|400|640` 参数时，表示先按照原始图片的 80% 缩小整张图片，再按照宽 400 像素高 640 像素以图片中心为基点进行图片裁剪；当客户单传递 `mark|aHR0cDovL2Js2cuY3Nkbi5uZXQveWlud2Vuamll->cut|800|640` 参数时，表示先在图片上加文字水印，文字信息为编码前的值，然后在按照宽 800 像素高 640 像素以图片中心为基点进行图片裁剪。

### 12.3.2 Redis 中的数据结构选择

在“存储什么样的数据”方面，初步来看有两种选择：一种是存储原始图片数据；另一种是存储经过各种图片处理器经过处理后的用户最终需要的图片数据。显然后者在客户体验性、需求契合度和存储效率上更为合适，而如果存储原始图片，那么不但存储单张图片需要的缓存容量更大，更关键的是这样的原始图片大多数情况下客户端并不需要。最后虽然缓存原始图片

可以降低减轻物理磁盘的 I/O 压力，但并不能减轻图片服务器上 CPU 的计算压力，因为图片服务器在从缓存系统中取出原始图片数据后，大多数情况下都会再根据客户端的要求进行各种图片特效运算，而这一部分操作非常消耗 CPU 资源。

根据以上的分析，我们可以在“存储什么样的数据”方面很快形成讨论结果。那“存储成哪种数据结构”方面又是怎样一个思考呢？最直观的判断是，既然图片服务器向 Redis Clusters 中读取的是经过各种特效处理后的图片效果，而一张原始图片根据不同的特效组合处理后，得到的效果也不一样。所以应该使用 Redis 中的简单 K-V 结构进行存储，其中的 Key 应该是原始图片的路径 + 客户端给定的特效处理参数，而 Value 则应该是经过处理后的图片数据。

但实际情况真是这样吗？实际情况是以上的内容描述并没有考虑太多性能方面的细节，这里我们至少还需要讨论一个重要性能点：数据文件的大小。虽然一个 128×128 像素大小 24KB 的文件数据，相对于物理介质上的存储来说算是一个小文件，但是它在单个 Redis 上的存储却属于一个大文件——我们一般在 Redis 上存储的缓存数据也不过是 1KB（例如一个经过序列化的用户信息）。而很多技术资料也表明当单个 Redis Value 的大小大于 10KB 时，Redis 对于这个 Value 的读写性能会大大降低，甚至一部分技术资料还给出了具体的数据，来说明写操作的性能测试结果。另外，Redis Cluster 保证性能的一个办法是在客户端将 Key 做一次 CRC16 运算，并根据计算结果将不同的 Key 送入不同的 Redis Cluster Master 节点，这样多个 Redis Cluster Client 就可以在同一时间完成多个 Key 的操作。Java 版本的 Jedis Client，其 CRC16 算法工具类的类名是 `redis.clients.util.JedisClusterCRC16`。

根据以上的分析，在存储一个 24KB 的图片文件时，我们不能直接将这个文件使用一个 K-V 结构存储到 Redis Cluster 的某一个节点上，而是应该将这个较大的数据文件分成若干数据段并对应不同的 Key，Key 的命名原则是能够通过 CRC16 算法，计算出不同的 Slot 目标结果。并且还应该将这个图片的 size 进行缓存，以便读取时使用。什么叫作让 CRC16 算法呈现不同结果呢？请看如下测试代码：

```
public static void main(String[] args) throws Exception {
    .....
    // 以下模拟某个图片在 Redis Cluster 存储的 3 个分段
    System.out.println(JedisClusterCRC16.getCRC16
("/path/imagename.png|1"));
    System.out.println(JedisClusterCRC16.getCRC16
("/path/imagename.png|2"));
    System.out.println(JedisClusterCRC16.getCRC16
("/path/imagename.png|3"));
    .....
}
```



```
// 以下是计算结果
6450
10577
14704
```

以上计算结果当 Redis Clusters 中 master 的节点个数大于 12 时，图片的 3 个分段就会被存储到两个 master 节点中（6450 slot 和 10577 slot 在一个节点上，14704 在另一个节点上）。在这个图片服务系统的示例中，我们将固定 5KB 为一个文件分片，也就是说以上举例的 24KB 图片文件数据会有 5 个文件分段。存储过程如图 12-8 所示。

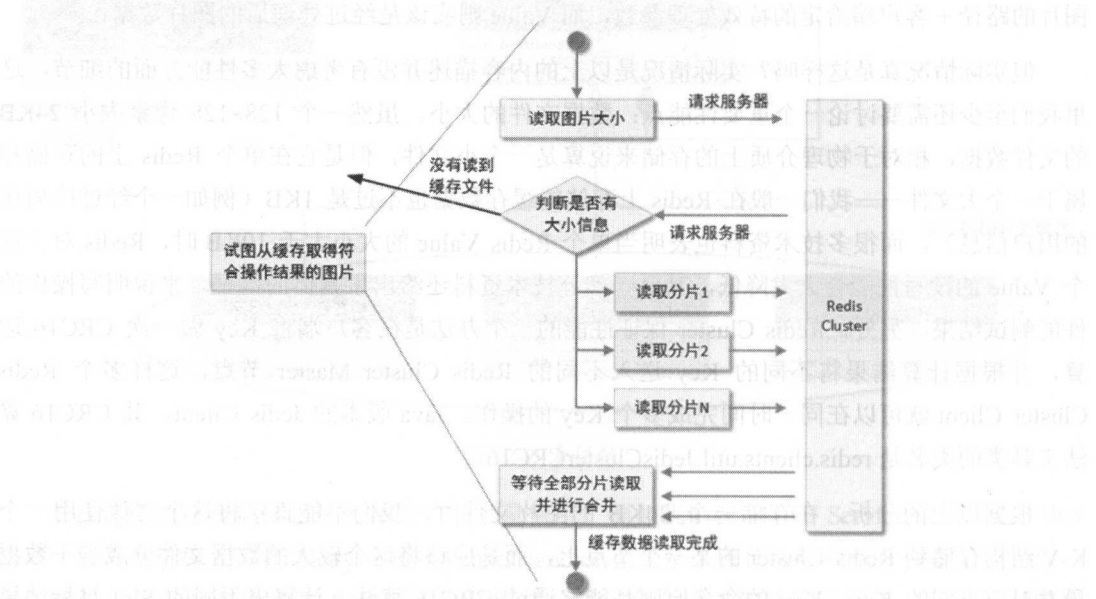


图 12-8 对同一张图片进行分段存储

当读取缓存文件时，客户端会首先去 Redis Cluster 上读取缓存图片的大小，以便重新确定文件有几个分片，然后再到 Redis Cluster 中读取每个数据分片。注意，虽然每个图片数据分片都设置了同样的过期时间，但由于每个节点的实际工作状态不同，所以还是可能出现某些分片数据读取失败，所以这个时候如果任何一个分片读取失败就认为整个读取过程失败。如果出现这样的情况，图片系统就会放弃缓存中的数据，并到下层的分布式文件系统上读取数据。

12.3.3 使用 Spring Boot

这个示例版本的图片服务工程将基于 Spring Boot 进行构建。Spring Boot 由 Pivotal 团队提供，它是基于 Spring Core 4.X 版本构建的一套组件库，既定目标是大幅度减少 Spring 工程在初始化搭建时的配置工作量。举个例子来说，在使用 Spring (3.X 版本尤为突出) 时，会产生

大量的 xml 配置信息，至少需要在配置信息中设定 ScanBase 的包路径、设定若干 ApplicationListener、配置数据库数据源、配置各种对象池和连接池。此外如果没有部署持续化集成服务，那还需要自行管理多套配置信息，以便将工程应用到不同的部署环境。

使用 Spring Boot 后，最直观的现象就是以前工程中的所有配置所需的 Spring XML 文件都消失了，Spring Boot 会按照“约定优于配置”的原则，自动对工程进行扫描并提炼出需要在工程启动是加载的 Bean、ApplicationListener、启动线程、预处理数据等资源。为了方便读者阅读下载的源代码，图 12-9 给出了整个工程的大致结构。

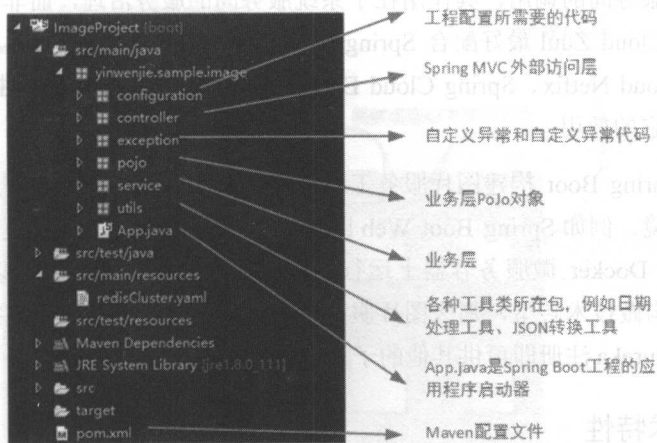
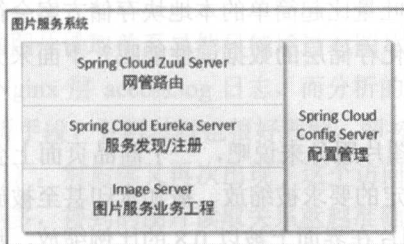


图 12-9 图片示例工程结构

你还可以直接基于 Spring Boot 为图片服务系统集成一套服务治理框架 Spring Cloud，但最终你会发现这样做又涉嫌过度设计——除非你的团队已经搭建了 Spring Cloud 基础应用环境，而图片服务只是作为一个服务提供节点注册到现有顶层系统中（图 12-10）。

涉嫌过度设计的图片系统



比较合理的子系统设计

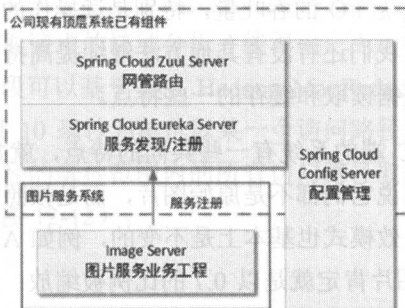


图 12-10 不要进行过度设计

相当一部分面向互联网的业务系统、图片服务系统虽然属于一种顶级子系统，但它必须依附于顶层系统的全局规划。而且图片服务系统对外暴露的服务接口太少，按照目前的功能规划，它对外暴露的服务接口就只包括三个：单一图片上传、批量上传和图片显示/下载。将为最顶层全局规划服务的 Spring Cloud Eureka Server（服务发现/注册）放置到图片服务系统中既没有必要也不合理。

在 12.2.3 节中，从技术功能的角度讨论了为什么图片服务的路由层采用 Nginx 而不是 Spring Cloud Zuul 的原因。这里我们再从系统结构层面进行以下补充：Spring Cloud Zuul 着眼于系统服务和系统服务间的调用，其作用在于系统服务间的服务治理，而非单一系统内部的调用；另外，Spring Cloud Zuul 最好配合 Spring Cloud 的其他组件进行使用，例如 Spring Cloud Security、Spring Cloud Netflix、Spring Cloud Eureka，而在单一系统内部单独使用 Spring Cloud Zuul 基本不能发挥它的效用。

最后，基于 Spring Boot 构建图片服务工程，也是为了后续的工程部署过程能够灵活选用运行环境和集成环境。例如 Spring Boot Web 比起传统的 Spring Web 工程更能简便地在微服务组件上运行，如在 Docker 微服务容器上运行；再例如，如果你所在公司以后决定向 Spring Cloud 服务治理架构做技术转型，那么图片服务也可以方便地进行升级，直接将自己提供的服务向 Spring Cloud Eureka 注册即可供其他的子系统服务使用。

## 12.3.4 其他技术特性

以上技术特性都是图片服务的第一个版本需要实现的，但在后续为了完善图片服务功能可以为图片服务增加一些新的技术特性。

### 1. 预读

本书介绍技术选型时提到，将用一款分布式文件系统作为对原始图片文件进行持久存储的技术选型，并且使用高性能的 SSD 固态硬盘 + 磁盘阵列作为分布式文件系统的物理层支持。这样一来系统 I/O 的吞吐量，特别是读操作的吞吐量比起简单的本地块存储方案会得到很高的提升。那么我们还有没有其他方法继续提高持久化存储层的数据读性能呢？下面来分析一下图片系统中数据读取和缓存的一些特点。

面向 C 端的系统有一些共同的特点，就拿图片服务来说吧，一个商品页面上会有很多图片，一般来说它们都不是原始图片，而是按照一定的要求被缩放、被加水印甚至被旋转的；它们使用的特效模式也基本上是不变的，例如 A 图片在界面上被以 0.8 的比例缩放，那么同一页面上的 B 图片肯定就是以 0.7 的比例被缩放；最后，由于它们会在物理磁盘上被同时读取，所以它们在各级缓存中存在的时间基本上也是一致的，当某张图片过期时其他图片也会过期。

我们可以使用预读的概念，将这些有读取关系的图片从文件系统上一次性读取出来，这样对文件系统的读操作效率优于一次只读取一个文件的操作效率。预读技术基于局部性原理，局部性原理是说计算机上某些相关部分的资源，都会存在于一个集中的区域，CPU 寄存器、内存地址、磁盘数据等，当某个资源 X 被处理时，和它临近的若干资源也即将被处理。这个概念可以被运用到图片处理上：如果某张页面上的图片 A 被读取时，那这张图片上的其他图片也将同时被读取（图 12-11）。

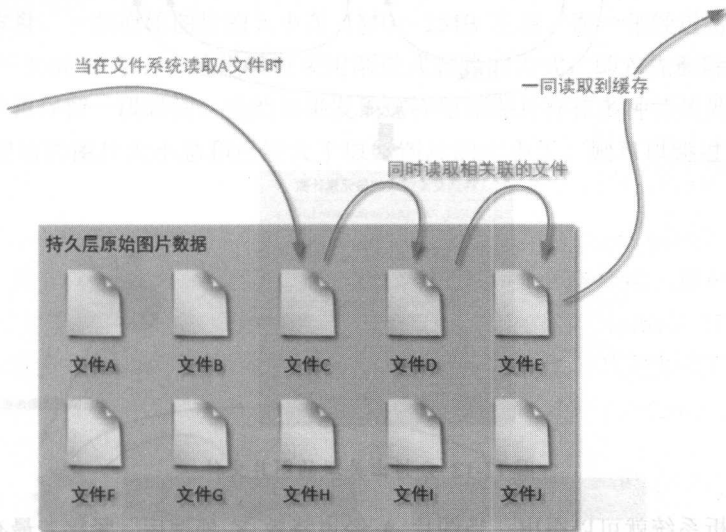


图 12-11 文件预读

由于图片系统并没有集成持久化数据库技术，所以无法记录某一个文件和哪些文件存在写操作联系。而且即使能够记录这些原始文件的上传关系，也不能作为文件预读的依据——因为客户端请求图片信息时并不是请求原始图片，而是请求经过特效处理后的图片，也就是说图片 C 经过特效处理后的图片 C1，和图片 D 经过特效处理后的图片 D1 才存在读取关联。

这样的图片读取关系显然只能通过对图片读取请求的持续分析才能得出，而这个分析源头可以基于 Nginx 层 access.log 日志，而分析的手段可以基于类似 Hadoop MapReduce 这样的离线/延迟分析手段。分析过程也很好理解，即按照 10 毫秒为单位以某一个访问路径为参照（带特效参数的），对后续又再次出现了这个访问路径的毫秒范围内的所有访问路径取交集，交集运算次数越多，得到的图片读取关系就越准确（图 12-12）。

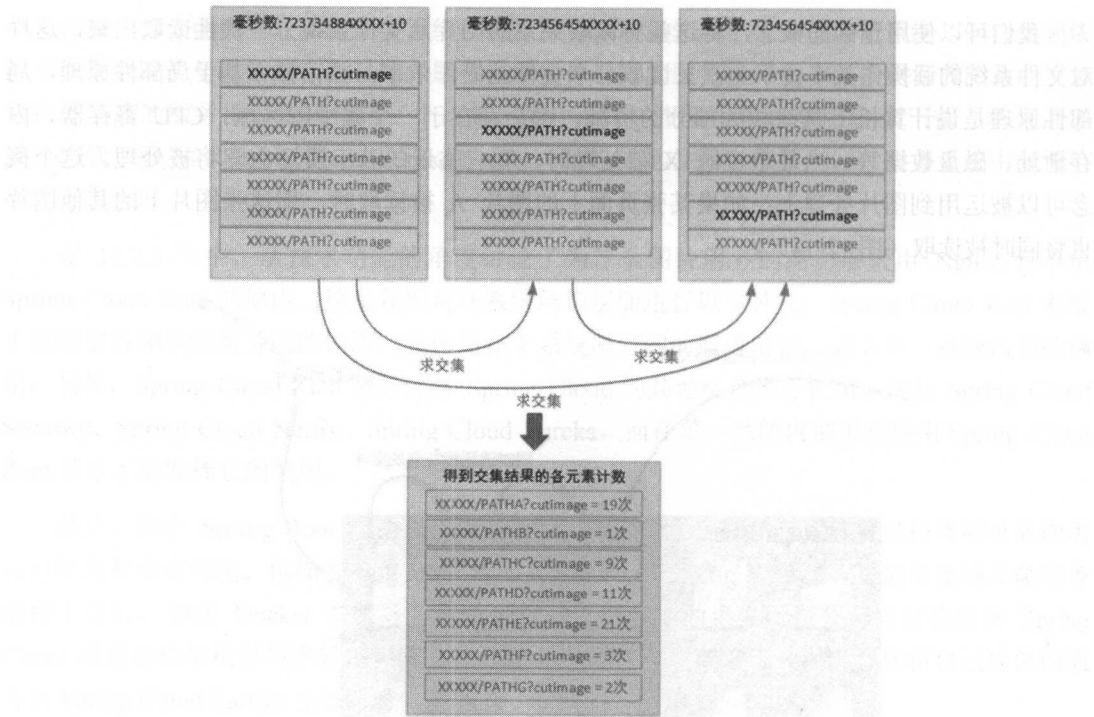


图 12-12 分析出关联的图片文件

这样数据分析系统就可以得出，当图片 A 经过特效 X 处理后，紧接着最有可能读取的其他文件和需要加载的特效，这样图片系统就可以对后续的图片进行预读并在完成特效处理存储到缓存系统中。这个图片关联关系的分析工作计算量比较大，以上只是计算的某一个文件的关联情况，试想一下所有的图片都要进行类似的分析过程，然后还要过滤出重复的分析数据，所以只有依靠大数据分析手段才能完成。

2. 图片删除

我们一直没有讨论过图片的删除问题，因为并不是所有的图片系统都需要图片删除功能，甚至有些系统还会特别说明所有的原始图片都要进行永久保存。但如果图片系统的存储容量确实有限，并且团队暂时没有太多资金进行存储扩容，那么删除一些不再使用的图片就是一个节约存储容量的好办法。但关键问题是，怎样判断图片不再使用呢？

最直观的思路是，按照图片上传时间向后推导 3 至 6 个月的时间到一个固定的时间点，如果超过这个固定时间点就将这张图片删除。但这样做的话图片系统并不能确定这些图片在后续的时间不会被请求者访问，例如一些畅销商品甚至会保持 1 年以上的销售热度。还有一种删除思路，是由客户端自行进行删除操作，例如当一个商品下架时同时删除商品图片。但这样做也



有问题，因为后续运营团队可能还会在进行后期销售总结时访问这个商品的快照信息，这时也会同时查看这个商品的图片。

怎样删除才是较合理的呢？首先是删除时机的问题，显然给定一个固定的时间长度作为删除依据是不满足要求的，时间长度的选择应该是动态的：利用数据分析系统得出当前某张图片最后一次访问时间，如果当前时间离该图片最后一次访问时间大于规定的阈值（例如3个月、6个月等值），就启动删除过程。另外从删除策略上来说，一张图片的删除不能不留余地直接删除原始图片本身，一张原始图片的大小在1MB~2MB左右，而一张经过特效处理后的图片大小在100KB~300KB左右。这里可以采用渐进式删除的方式：即首先删除原始图片，对特效处理后的图片再保留一段时间。当然如果发现这样原始图片存在多种特效处理规则，并且经过这些规则处理后的图片大小总和已经大于原始图片的大小了，则可以跳过渐进式删除过程（图12-13）。

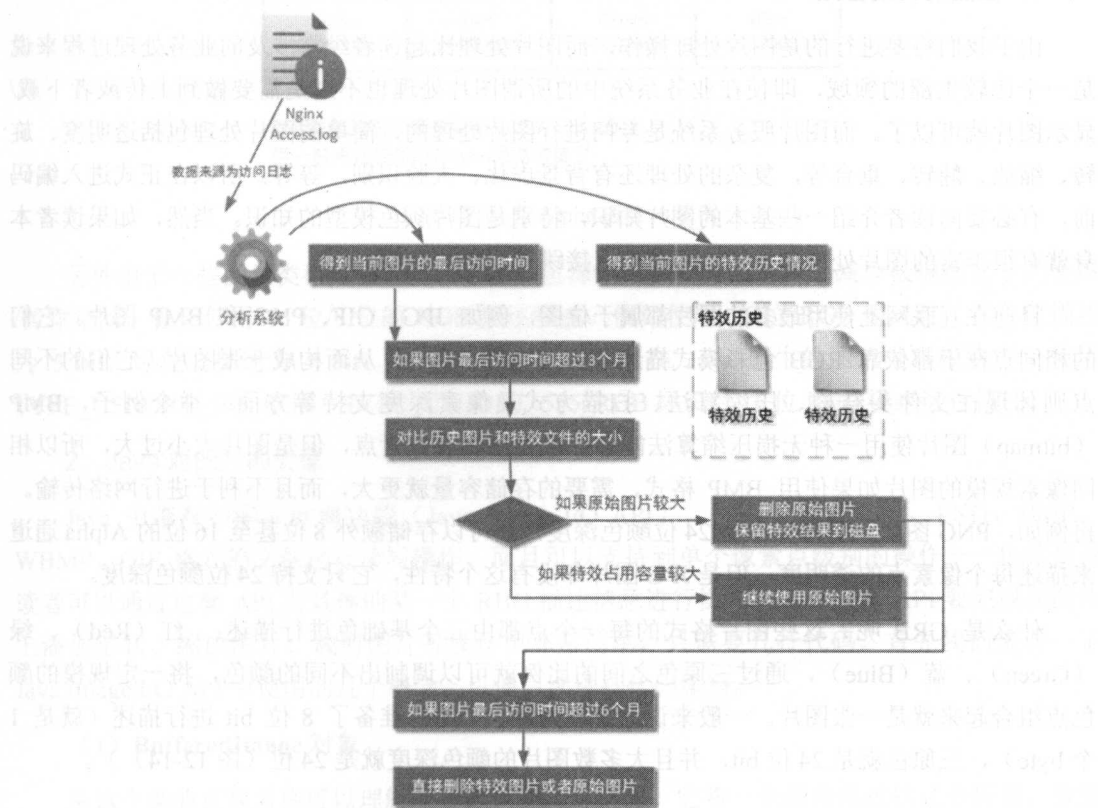


图 12-13 删除图片文件

通过删除原始图片保留特效结果文件的方式，可以有效防止原始图片删除后用户零星访问

的空窗期。待到一个更长的，再无任何图片访问请求的时间后，最终将图片所有的存储痕迹全部抹去，这时如果用户再进行访问就会出现图片已过期的提示。通过渐进式删除过程，一般可以在删除的第一阶段腾出 20%~30%左右的存储容量，而且不会对用户后续的零星访问造成任何影响（但不在允许用户设定新的特效了），最后在保证用户有 90%以上的几率没有再次访问该图片的可能后，再对图片进行正式删除。渐进式删除不适合所有的图片服务，笔者还是建议在存储容量充足、集群服务性能足够的情况下对原始图片进行永久保存（至少 3~5 年）。

## 12.4 详细设计阶段

### 12.4.1 位图基本知识

#### 1. 位图的构成基础

由于我们将要进行的是图片处理操作，而图片处理比起读者经常涉及的业务处理过程来说是一个比较生涩的领域，即使在业务系统中的所谓图片处理也不过是需要做到上传或者下载/显示图片就可以了。而图片服务系统是专门进行图片处理的，简单的图片处理包括透明度、旋转、缩放、翻转、重合等，复杂的处理还有背景虚化、人脸识别，等等。所以在正式进入编码前，有必要向读者介绍一些基本的图片知识，特别是图片颜色模型的知识。当然，如果读者本身就很丰富的图片处理知识了，则可以直接跳过本节的介绍。

目前在互联网上使用最多的图片都属于位图，例如 JPG、GIF、PNG 和 BMP 图片。它们的相同点在于都依靠 RGB 色彩模式描述图片中的每一个点，从而构成一张图片。它们的不同点则体现在文件头结构、压缩算法、扫描方式及像素深度支持等方面。举个例子，BMP（bitmap）图片使用一种无损压缩算法能够保证还原所有像素点，但是图片大小过大，所以相同像素规模的图片如果使用 BMP 格式，需要的存储容量就更大，而且不利于进行网络传输。再例如，PNG 图片除了可以存储 24 位颜色深度，还可以存储额外 8 位甚至 16 位的 Alpha 通道来描述每个像素点的透明度，但是 JPG 图片却没有这个特性，它只支持 24 位颜色深度。

什么是 GRB 呢？这些图片格式的每一个点都由三个基础色进行描述：红（Red）、绿（Green）、蓝（Blue），通过三原色之间的比例就可以调制出不同的颜色，将一定规模的颜色点组合起来就是一张图片。一般来说计算机为每一个原色准备了 8 位 bit 进行描述（就是 1 个 byte），三原色就是 24 位 bit，并且大多数图片的颜色深度就是 24 位（图 12-14）。

Red 00000000	Green 00000000	Blue 00000000
-----------------	-------------------	------------------

8位二进制最多可以存储0—255的二进制数值  
十六进制数值记录为0—FF

图 12-14 位图结构

这样看来 24 位颜色深度的一个图片点，最多可以记录  $256 \times 256 \times 256 = 16777216$  种颜色。那么类似 PNG 图片使用的 32 位/48 位颜色深度有代表什么意思呢？多出来的 8 位/16 位为了记录这个点的可见度（透明度），这个记录范围的数值又称为 Alpha 通道。这样同一个颜色配合不同的可见度就可以满足更丰富的颜色展示要求。另外，这也是 JPG 文件不能支持透明度的原因——它的格式规范中不带有对 Alpha 通道的支持（图 12-15）。

Alpha 00000000	Red 00000000	Green 00000000	Blue 00000000
-------------------	-----------------	-------------------	------------------

PNG类型的图片最多支持16位的Alpha通过，  
但一般来说使用8位Alpha通道就够了

图 12-15 带有 Alpha 通道的位图

另外由于一些图片类型只需要记录更浅的位深，所以一些图片为了减少极端情况下的使用/传输空间，也会使用 16 位/8 位的 RGB 描述模式。例如使用 16 位 RGB 模式时，红色位描述为 5 bit、绿色位描述为 6 bit、蓝色位描述为 5 bit。在 Java 原生的图片处理模块中，使用 TYPE\_USHORT\_565\_RGB、TYPE\_USHORT\_555\_RGB 标识对 16 位 RGB 模式进行描述。

2. Java 对图片的处理

Java 中自带的图片处理功能（Java Image I/O API），能够支持对 JPG、PNG、BMP、WBMP、GIF 格式的文件进行读写操作，而且可以支持到单个像素点级别的操作——也就是说读者可以通过这套 API 对具体的某一个 RGB 描述信息进行操作。通过这套 API 要完成在图片上添加形状、缩放图片、裁剪图片等操作也非常简单，只需要几行代码。首先我们来看一下 Java Image I/O API 中使用的几个概念，以便后续进行代码编写。

(1) BufferedImage 对象

从这个类的直观名称可以理解成被缓存的图片信息，它将一张图片经过格式分析后，放置在内存中的一个可访问区域。开发人员可以从这个可访问区域提取到很多关于这张图片有用的信息，例如可以取得 ColorModel 表示的颜色分量，里面包括了每一个像素的 RGB 信息和

Alpha 信息；还可以取得 `Raster` 表示的像素矩阵，通过它可以读取一个范围内的像素点。开发人员对这个区域进行读写操作，实际上就是对这张图片进行像素级别操作。以下代码可以加载一张位图到 `BufferedImage` 中：

```
.....
BufferedImage srcImage = javax.imageio.Imageio.read(new File
("mmexport1444022819048.jpg"));
.....
```

要注意的是，`BufferedImage` 一旦被加载其像素规模就不能改变了，例如你最初加载了一个  $800 \times 600$  像素规模的 JPG 文件到 `BufferedImage` 中，在操作过程中就不能将这个 `BufferedImage` 缩小成  $600 \times 400$  像素的。如果要这样做，那只能创建一个新的 `BufferedImage`，并将进行缩放后的计算结果加载到这个新的 `BufferedImage` 中。以下的方式可以创建一个空的 `BufferedImage`：

```
.....
// 以下代码创建一个  $600 \times 400$  像素的 BufferedImage
BufferedImage outputImage = new BufferedImage(600, 400, BufferedImage.TYPE_
INT_RGB);
.....
```

请注意以上代码片段中的最后一个参数 `BufferedImage.TYPE_INT_RGB`，它指定了 `BufferedImage` 需要支持的 RGB 规则。`TYPE_INT_RGB` 代表了使用一个 `int` 数值表示 24 位深度的 RGB 规则，`TYPE_USHORT_565_RGB` 表示，使用一个 `short` 数值，表示 16 位深度的 RGB 规则，其中红色占 5 位，绿色占 6 位，蓝色占 5 位。再例如 `TYPE_INT_ARGB` 和 `TYPE_4BYTE_ABGR` 分别表示以 `int` 数值或一个 4 位的 `byte` 数组表示 ARGB 规则，换句话说就是这个 `BufferedImage` 支持图片透明度的表示。

最直观的理解就是，`BufferedImage` 相当于在内存区域中的画布，这个画布上可以有一张或者多张图片，也可以没有任何图片。你可以在画布上对每个像素进行读写操作，但是你不能改变画布的大小。

## (2) Graphics 对象

`Graphics` 类抽象一点说是进行图形操作的上下文控制的类，具体一点说是图形画笔工具。通过这个类（以及它的子类）开发人员可以方便地在画布上绘制不同的规则形状、图片或者文字。

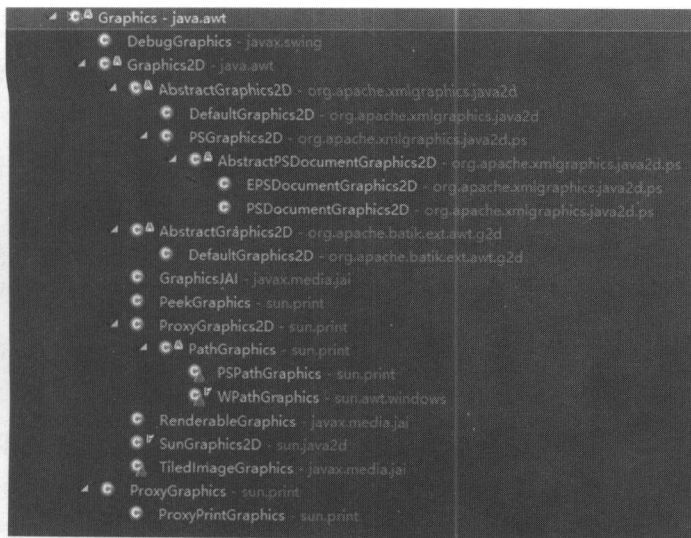


图 12-16 Graphics 类结构

图 12-16 中呈现的 Graphics 子类结构可以支持不同的绘图场景，例如 Graphics2D 类基本上可以处理所有主流二维位图的画布绘制，在我们的图片服务系统中使用最多的画笔类也是它。以下代码可以在画布上绘制一个规则的矩形，并填充颜色：

```
.....
// 创建一个 100×80 像素的画布
BufferedImage outputImage=newBufferedImage(100, 80, BufferedImage.TYPE_
INT_RGB);
// 获得这个画布的画笔，也可以使用 createGraphics 创建画笔
Graphics graphics = outputImage.getGraphics();
// 从画布上 10,10 的坐标开始，绘制一个 60×40 像素的矩形
graphics.setColor(Color.RED);
graphics.drawRect(10, 10, 60, 40);
// 处理
graphics.dispose();
.....
```

通过以上代码“红色矩形”就在画布上被绘制出来了，但通过观察运行结果可以发现画布的底色是黑色，这是为什么呢？这是因为我们使用的 BufferedImage 构造方法将使用 RGB0000 0000 0000 0000 0000 0000 的数值进行每个像素点的初始化，实际上就是黑色的 RGB 值。我们换一种方式进行 BufferedImage 的初始化，就可以将 BufferedImage 中的像素点初始化成白色：

```
.....
int width = 100, height = 80;
int size = width * height;
```



```

int[] pixels = new int[size];
// 现在设置每一个像素点的 RGB 值为白色
for(int index = 0 ; index < size ; index++) {
    pixels[index] = 0xFFFFFF;
}
// size 就是像素规模大小
DataBuffer dataBuffer = new DataBufferInt(pixels, size);
// 初始化的 Raster 类, 就是像素矩形数组的封装
WritableRaster raster = Raster.createPackedRaster(dataBuffer, width,
height, width, new int [] { 0xFF0000, 0xFF00, 0xFF }, null );
DirectColorModel directColorModel = new DirectColorModel(24, 0xFF0000,
0xFF00, 0xFF);
// 生成 BufferedImage, 这样 BufferedImage 中的每个像素就是白色了
BufferedImage outputImage = new BufferedImage(directColorModel, raster,
true , null );
.....

```

### 3. JVM 进行针对性优化

图片处理操作是计算密集型操作, 非常消耗 CPU 资源和内存资源。而 Java Image I/O API 又是基于 Java 进行的图片像素级操作, 其处理性能本身就不及 C/C++。所以对 JVM 的内存优化就显得非常重要了。这里笔者假设读者已经知道了 JVM 的基本原理和当前最流行的 JVM Hotspot 模型, 所以笔者直接讲解 JVM 的几个优化注意点。

(1) 关于-Xmx 最大堆内存: 由于我们假设的图片处理场景是一个百万级 PV 的中等电商平台, 所以单个图片服务系统单位时间内需要处理的图片请求数量也是比较大的, 首先建议改变-Xmx 参数, 设置内存大小在 8GB 以上或者设置内存大小为操作系统可用内存的 60%以上。注意也不能太大了, 这要依据你的 CPU 性能设定, 否则就可能会出现 full gc 明显卡顿现象。

(2) 关于回收器的选择: 回收器的设置是最关键的, 我们知道在早期的 JDK 版本中提供的回收器, 都是采用中断用户线程的方式进行, 无论是针对新生代的 Serial、ParNew 还是针对年老代的 SerialOld, 都是这样。但在高并发环境下, 如果出现用户线程的停顿现象, 就会在非常短的时间内造成大量请求等待, 严重影响服务器的处理效率。所以对 JVM 的优化要特别注意这个点, 建议采用 JDK 1.7+ 64 位以上的运行版本, 并设置 JVM 到 Server 模式。

首先是指定年老代使用的回收器, 这个推荐使用标记—清除算法的 CMS (响应时间优先回收器) 就好了, 它会在尽量保证用户线程运行的情况下对待回收区域进行多次标记回收。还要注意, CMS 回收器并不能保证在回收时用户线程绝对不停止, 而是使用两次短暂的挂起操作取代之之前回收器使用的一次较长的挂起操作。另外注意, CMS 的线程数量有一个默认值, 这个默认值是 (CPU 内核数量 + 3) / 4。虽然这个数量是可以设置的, 但是笔者并不建议读者

自己去设定这个值，而是保证你的操作系统上至少有 8 个或以上（16 个最佳）CPU 内核数量（如果达到或超过 24 核，就要通过 `-XX:ParallelGCThreads` 参数控制一下了），`-XX:+UseConcMarkSweepGC` 参数可开启 CMS。

接着是新生代使用的回收器，如果你指定了年老代使用回收器为 CMS，那么新生代的回收器就不能使用 `ParallelScavenge` 回收器了（吞吐量优先回收器），因为两种回收器不兼容。首先 CMS 也支持使用 `ParNew` 回收器，这是一个单线程 `Serial GC` 的一个多线程版本，虽然在进行 GC 时会出现用户线程挂起的情况，但由于它是多线程的版本，且我们存储的数据特点决定了 `ParNew` 不会出现太大的性能瓶颈。

（3）关于年老代和新生代的比例问题：虽然我们常说 JVM 优化的目的减少移动到年老代的对象数量和减少 full gc 的情况，但由于我们在内存中将要存储和操作的数据比较特别——图片文件数据，形象来说是 `BufferedImage` 对象、较大的 `byte[]` 数组对象。这首先些对象的特点是数据量较大，一张原始图片小则 500KB，大则会达到 1MB（不会再大了，因为我们对上传文件的大小做了限制）。其次图片处理速度较慢，一张颜色深度在 24 位大小在 1MB 的 JPG 文件，等比例缩放成一张 200KB 的图片耗费的时间在 200ms 左右，甚至有时会超过 500ms。所以这样的数据存放在新生代区域，就会造成非常频繁的复制操作和向年老代的移动操作。这样我们的优化思路就需要有针对性：减少新生代的内存空间，并设置一个阈值，在图片数据超过这个阈值时就直接将对象放入年老代，然后在年老代中由 CMS GC 进行回收。例如当 JVM 堆内存数量为 8GB 时，可以通过 `-Xmn` 参数设置新生代（eden+ 2 survivor space）的空间大小为 1GB；通过 `-XX:PretenureSizeThreshold=<byte size>` 参数设置直接进入年老代的对象大小值；通过 `-XX:CMSInitiatingOccupancyFraction=50` 参数设定当年老代已使用的内存大小达到 50% 时，开始 CMS GC。

（4）关于持久代的问题：在我们的图片服务中，JVM 的持久代并不会存储太多数据，特别是我们的工程中需要加载 IOC 容器的 class 信息并不多，且常量信息也不多的情况下。况且在 JDK Version 1.8+ 的版本中 JVM 模型已经取消了对持久代的支持。所以持久代并不需要太多针对性的优化，最后 JDK Version 1.8+ 也是笔者推荐使用的。

#### 4. 其他说明

除了 Java 提供的 `Java Image I/O API`，还有一些基于 Java 开发的第三方图形组件，但是很多第三方图形组件已经没有再维护了，如果各位读者有兴趣可以进行研究：例如 `Java Image Filters`、`JMagick` 等。

Java 提供的 `Java Image I/O API` 图片处理工具虽然可以进行像素级别的操作，但是相对于 C/C++ 提供的图片处理性能来说还是较弱，那么要进行图形高效运算的语言基础还是 C/C++ 为

宜。目前流行的 2D 和 3D 图像处理软件也多是基于 C/C++ 构建, 例如 OpenGL、DirectX 等。另外图形处理都是 CPU 密集型工作, 对计算机的运算资源和内存资源要求都比较高, 目前的发展趋势是使用专门的 GPU 代替 CPU 进行运算, 这也是为什么无论是我们使用 Java Image I/O API 还是基于 Nginx 的 Image 模块为系统提供简单的图片处理功能, 对 CPU 要求都非常高的原因。

## 12.4.2 Nginx 中的 Proxy Cache 配置

根据前文对图片服务的系统架构规划, Nginx 充当了第三级缓存的作用和对图片请求服务的负载均衡作用, 所以在 Nginx 配置文件中, 至少需要对 Nginx Proxy Cache 功能和 UpStream 功能进行配置。以下为主要的 Rewrite 部分和负载均衡部分的配置:

```
.....
upstream imageserver {
    server 192.168.61.1:8080;
    server 192.168.61.2:8080;
    #建议配置健康检查部分
}
.....
server {
    .....
    location /imageQuery {
        # 后续还要增加 proxy cache 部分的配置
        # cluster loader
        proxy_pass http://imageserver;
    }
    .....
}
```

请记得按照我们在本书第 3 章中对 Nginx 优化细节的讨论, 进行其他配置信息的调整工作。以下配置信息是和 Proxy Cache 相关的配置信息:

```
.....
proxy_cache_path /nginxcache/imagecache levels=1:2 keys_zone=imagecache:
500m inactive=300s max_size=1g;
server {
    .....
    # 带有特效参数的 URL 格式适配
    location ~* ^/image/(.*)\.(jpg|jpeg|png|gif)&(.*)$ {
        # rewrite
        rewrite
        ^/image/(.*)\.(jpg|jpeg|png|gif)&(.*)$ /imageQuery/$1.$2?special=$3 last;
    }
}
```

```

# 不带有特效参数的 URL 格式适配
location ~* ^/image/(.*)\.(jpg|jpeg|png|gif)$ {
    # rewrite
    rewrite ^/image/(.*)\.(jpg|jpeg|png|gif)$ /imageQuery/$1.$2 last;
}
# 重写后的访问路径
location /imageQuery {
    # proxy cache
    proxy_cache imagecache;
    proxy_cache_valid 200 304 300s;
    proxy_cache_key $uri$query_string;
    # cluster loader
    proxy_pass http://imageserver;
}
.....
}

```

为了便于客户端拼凑字符串，图片服务系统向客户端提供了一个约定俗成的 URL 地址结构，客户端可以在原始图片的 URL 后面使用“&”符号给出要求处理的图片特效，类似如下结构：

```

http://imagenginxproxy/image/20170114/e1733711-8e4d-4d9e-b230-
dd22898313ef.png&zoomimage%7Cratio%3D0.9-
%3Emarkimage%7CmarkValue%3Dwww.yinwenjie.net111

```

当 Nginx 收到这个请求后，将会重写（rewrite）这个 URL 到图片服务层的真实 URL 地址，并且将得到的数据通过 Proxy Cache 进行缓存。关于 upstream 功能的配置和 rewrite 功能的配置，我们已经在负载均衡专题中进行了讲解，这里就不再赘述了。这里主要再说明一下 Proxy Cache 的一些基本配置（关于 Proxy Cache 模块的更多配置项，可参考官方文档：[http://nginx.org/en/docs/http/nginx\\_http\\_proxy\\_module.html](http://nginx.org/en/docs/http/nginx_http_proxy_module.html)）：

- **proxy\_cache\_path**：这个参数指定 Nginx Proxy Cache 的主要配置项目，其中“/nginxcache/imagecache”表示 Proxy Cache 存储数据的主要目录，注意这个目录必须已经存在，否则启动时会报错；levels 参数指定 Hash 结构层次，例如“levels=1:2”表示有两级 hash 目录结构，其中第一级目录结构有一个字符，第二级目录结构有两个字符；keys\_zone 参数描述了 proxy cache 的名称和最大可使用的内存大小，这个名称在 Nginx 节点中必须是唯一的，而内存大小视实际情况进行设定，例如这里设置的就是 500MB；inactive 参数是指当数据持续多长时间没有被访问后，就删除这个缓存数据，这里设置“inactive=300s”表示 300 秒的过期时间，注意这个过期时间不要超过 Redis 中数据的过期时间，否则 Redis 的数据缓存就不起作用了，并且这个时间需要和 proxy\_cache\_valid 的时间区分开来。max\_size 参数是指 Proxy Cache 可以使用的最大磁



盘空间，例如设置成“max\_size=1g”，表示最大可以使用 1GB 的硬盘空间，很明显这个空间在正式环境下还应该加大。

- **proxy\_cache\_valid**: 该参数可以为不同的 HTTP 响应码，设置不同的缓存时间。例如在以上的配置信息中设置了“proxy\_cache\_valid 200 304 300s”表示当 Nginx 反向代理收到的 HTTP 响应码为 200 或者 304 时，才进行数据缓存，缓存时间为 300 秒——无论这 300 秒的时间内是否有请求，都是缓存 300 秒，这个就和 proxy\_cache\_path 设置项中的 inactive 参数的意义就不一样了。你还可以使用多个 proxy\_cache\_valid 为不同的 HTTP 响应码设置不同的缓存过期时间，例如：

```
.....
# HTTP 200 和 302 的缓存时间为 5 分钟
proxy_cache_valid 200 302 5m;
# HTTP 404 的缓存时间为 4 分钟
proxy_cache_valid 404 4m;
.....
```

- **proxy\_cache\_key**: Nginx Proxy Cache 模块缓存数据的原理，是建立 K-V 的映射关系。使用这个参数可以设定 Proxy Cache 模块计算 Key 的依据。例如下面的设置，就是使用 HTTP 请求的 URL 地址（不带参数）作为 Key 的计算依据：

```
proxy_cache_key $uri
```

但是作为图片服务的上层缓存模块来说，我们不能这样进行设置。这是因为同一张图片的不同特效要求，其 URL 地址都是一样的，只是参数不一样而已。否则同一张图片的不同显示效果，将会被 Nginx 缓存视为统一返回数据：

```
由于只以 URL 作为计算 Key 的依据，所以以下两个 URL 代表的特效，将会出现缓存错误
/imageQuery/20170114/e1733711-8e4d-4d9e-b230-
dd22898313ef.png?special=zoomimage|ratio=0.55
/imageQuery/20170114/e1733711-8e4d-4d9e-b230-
dd22898313ef.png?special=zoomimage|ratio=0.9
```

正确的设置方式，是必须指定 Proxy Cache 模块在进行 Key 计算时，要参考 URL 的参数设置。所以以下设置方式才是正确的：

```
.....
#其中$query_string, 就代表 URL 中的参数信息部分
proxy_cache_key $uri$query_string;
.....
```

### 12.4.3 责任链进行图片处理

在这个图片服务示例中，我们将基于责任链模式以生产线的方式，进行客户端要求的特效



处理过程。本小节我们对这部分主要的代码进行说明。在示例的工程中，我们已经实现了三个图片特效：图片等比例缩放操作、图片裁剪操作和字符串性质的水印操作（图 12-17）。

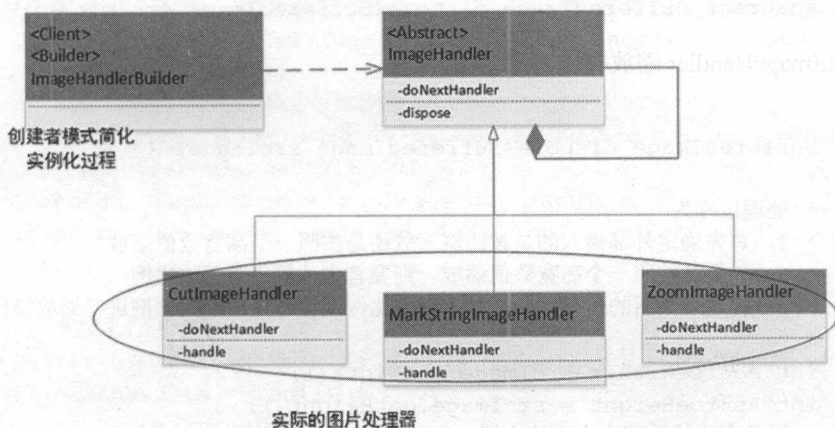


图 12-17 责任链模式

以上给出的类图中，除了前文介绍的责任链模式，还有一个创建者用来简化整个生产线的构建过程。创建者模式在很多组件中都有使用，例如 Protobuf 中创建对象所使用的手段就是先创建一个创建者，然后再由这个创建者进行实际对象的创建，示例如下：

```

.....
// Protobuf 中的实力创建
MessageS.Message.Builder messageBuilder = MessageS.Message.newBuilder();
// 用户名
messageBuilder.setUserName(username);
// 商品 id
messageBuilder.setBusinessCode(buid);
// 等一系列其他属性值
.....
// 再进行实际对象的创建
MessageS.Message messagePB = messageBuilder.build();
.....

```

以下是 ImageHandler 抽象类的部分定义，和 ZoomImageHandler 图片缩放处理器的重要代码片段。

#### • dispose 抽象方法定义

```

/**
 * 这个 dispose 方法，就是子类需要主要实现的方式
 * 如果处理过程中，不需要变更画布的尺寸，则可以在处理后将 srcImage 代表的画布直接返回
 * 如果在处理过程中，需要变更画布尺寸，则可以在实现的方法中创建一个新的画布，并进行返回
 */

```

```

* @param srcImage 从上一个处理器传来的处理过得图片信息（画布信息）
* @return 处理完成后一定要返回一个画布
*/
public abstract BufferedImage dispose(BufferedImage srcImage);

```

• ZoomImageHandler 缩放操作实现:

```

.....
public BufferedImage dispose(BufferedImage srcImage) {
    /*
    * 处理过程为
    * 1. 首先确定外部输入的是按比例缩放还是按照一个高宽数值缩放
    * 2. 如果是按照一个高宽数值缩放，则要首先计算一个缩放比例
    * 3. 构建一个新的画布，并按照指定的比例或者计算出来的比例进行缩放操作
    * */
    int sourceWidth = srcImage.getWidth();
    int sourceHeight = srcImage.getHeight();
    //得到合适的压缩大小，按比例
    int localDestWidth, localDestHeight;
    // 如果条件成立，则说明是按照比例缩小
    if(ratio != -1) {
        localDestWidth = Math.round((sourceWidth * ratio));
        localDestHeight = Math.round((sourceHeight * ratio));
    }
    // 否则是按照输入的宽、高重新计算一个比例，再进行缩小
    else {
        float localRatio;
        // 如果发现输入的目标高宽大于图片的原始高宽，则按照 ratio==1 处理
        if(sourceWidth <= this.destWidth || sourceHeight <= this.
destHeight) {
            localDestHeight = sourceHeight;
            localDestWidth = sourceWidth;
        }
        // 按照高计算
        else if(sourceWidth > sourceHeight) {
            localRatio = new BigDecimal(this.destHeight).divide
(new BigDecimal(sourceHeight), 2, RoundingMode.HALF_UP).floatValue();
            localDestHeight = (int)(sourceHeight * localRatio);
            localDestWidth = (int)(sourceWidth * localRatio);
        }
        // 否则按照宽计算
        else {
            localRatio = new BigDecimal(this.destWidth).divide
(new BigDecimal(sourceWidth), 2, RoundingMode.HALF_UP).floatValue();
            localDestHeight = (int)(sourceHeight * localRatio);

```

```

        localDestWidth = (int)(sourceWidth * localRatio);
    }
}
// 快速缩放算法
Image destImage = srcImage.getScaledInstance(localDestWidth, localDestHeight, Image.SCALE_FAST);
// RGB 位深为 24 位, 适合互联网显示
BufferedImage outputImage = new BufferedImage(localDestWidth, localDestHeight, BufferedImage.TYPE_INT_RGB);
Graphics graphics = outputImage.getGraphics();
graphics.drawImage(destImage, 0, 0, null);
graphics.dispose();

// 继续进行下一个处理
BufferedImage nextResults = this.doNextHandler(outputImage);
if(nextResults == null) {
    return outputImage;
}
return nextResults;
}
.....

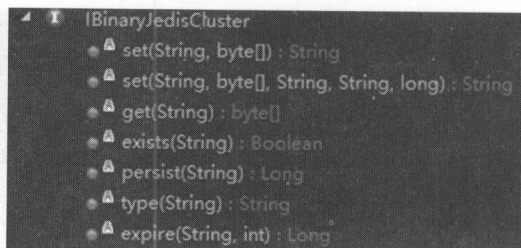
```

## 12.4.4 Redis 缓存操作

### 1. Redis Cluster Client 对 byte[] 操作

在这个示例的图片服务系统中, 对于 Redis 的操作有两个关键点。首先 Redis 官方推荐的 Java 客户端实现 Jedis, 提供了一个 `redis.clients.jedis.JedisCluster` 类对 Redis Cluster 进行操作。问题是这里面只提供了针对 String 类型的 Value 进行操作, 读者可以查看 JedisCluster 类的源代码进行验证。还好 Jedis 中的这部分代码不难懂, 那么要满足我们图片服务系统中对 byte 类型的 Value 进行操作的要求, 我们可以仿照 JedisCluster 类的实现自行实现一个, 分别是 `IBinaryJedisCluster` 接口和 `BinaryJedisClusterImpl` 实现。这里的代码不需要按照 JedisCluster 类中实现对 Redis 中所有数据接口的操作, 只需要实现我们系统中需要的 Redis 操作即可。

- 如图 12-18 所示是 `IBinaryJedisCluster` 接口。



```

IBinaryJedisCluster
    • set(String, byte[]) : String
    • set(String, byte[], String, String, long) : String
    • get(String) : byte[]
    • exists(String) : Boolean
    • persist(String) : Long
    • type(String) : String
    • expire(String, int) : Long
  
```

图 12-18 IBinaryJedisCluster 接口

- 以下是 BinaryJedisClusterImpl 类中的部分实现代码:

```

.....
/**
 * Redis Cluster 客户端操作实现, 这个类参考自 JedisCluster
 * 源自 JedisCluster 对 JedisClusterCommand 的封装只能对 String 类型的 Value 进行
 * 处理
 * 不能对 byte[]形式的 Value 进行处理
 * @author yinwenjie
 */
public class BinaryJedisClusterImpl implements IBinaryJedisCluster, Closeable {
    public static final short HASHSLOTS = 16384;
    private static final int DEFAULT_TIMEOUT = 2000;
    private static final int DEFAULT_MAX_REDIRECTIONS = 20;
    .....
    @Override
    public String set(final String key, final byte[] value, final String
    nxxx, final String expx, final long time) {
        return new JedisClusterCommand<String>(connectionHandler, maxRedirect
    tions) {
            @Override
            public String execute(Jedis connection) {
                return connection.set(key.getBytes(), value, nxxx.getBytes(),
    expx.getBytes(), time);
            }
        }.run(key);
    }
    @Override
    public byte[] get(final String key) {
        return new JedisClusterCommand<byte[]>(connectionHandler,
    maxRedirections) {
            @Override
            public byte[] execute(Jedis connection) {
                return connection.get(key.getBytes());
            }
        }.run(key);
    }
}
  
```



```

        }.run(key);
    }
    .....
    @Override
    public Boolean exists(final String key) {
        return new JedisClusterCommand<Boolean>(connectionHandler,
maxRedirections) {
            @Override
            public Boolean execute(Jedis connection) {
                return connection.exists(key);
            }
        }.run(key);
    }
    .....
}

```

## 2. 保证 Key 的数据在 5KB 以内

在 12.3.2 节，还介绍到 Redis 对超过 10KB 的 Value 信息有较高的读写延迟，而我们存储到 Redis 中的图片信息又都比较大（20KB 是普遍现象，200KB 也是可能的），在设计章节中提到的处理办法是将一个图片信息拆分为多个 Value 存储到 Redis Cluster 中的不同节点上。以下是进行 Value 存储时的主要代码片段：

```

    .....
    @Override
    public void saveCache(String imageURL, byte[] imagebytes) throws
BusinessException {
        /*
         * 1. 经过合法性验证后，处理的第一步就是判断 imagebytes 需要被分成几个段
         * 2. 然后进行 byte 的拆分和保存
         * 3. 只有所有的段都保存成功了，才进行返回（同步的，连续性的）
         */
        if(StringUtils.isEmpty(imageURL) || imagebytes == null) {
            throw new BusinessException(" 参 数 错 误 ， 请 检 查 ! ",
BusinessCode._404);
        }
        //确定分段
        Integer maxLen = imagebytes.length;
        Integer splitNum = maxLen / IImageEffectsCacheService.PERPATCH_
IMAGE_SIZE;
        Integer remainLen = maxLen % IImageEffectsCacheService.PERPATCH_
IMAGE_SIZE;
        if(remainLen != 0) {
            splitNum++;
        }
    }
}

```



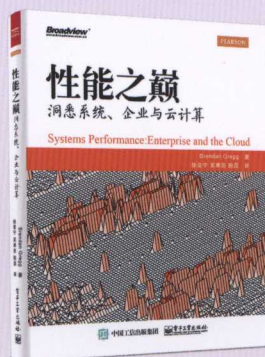
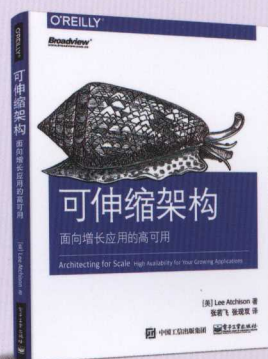
```

    }
    // 开始构造 key
    String key[] = new String[splitNum];
    for(int index = 0 ; index < splitNum ; index++) {
        key[index] = imageURL + "|" + index;
    }
    String lenkey = imageURL + "|size";
    .....
    // 保存长度信息到缓存系统
    binaryJedisCluster.set(lenkey, maxLen.toString().getBytes());
    for(int index = 0 ; index < splitNum ; index++) {
        byte[] values = null;
        Integer valuesLen = null;
        // 确定本次要添加的分片长度
        if(index + 1 == splitNum) {
            valuesLen=remainLen==0?IImageEffectsCacheService. PERPATCH_IMAGE
_SIZE:remainLen;
        } else {
            valuesLen = IImageEffectsCacheService.PERPATCH_IMAGE_SIZE;
        }
        values = new byte[valuesLen];
        // 复制 byte 信息
        // System.arraycopy 是操作系统级别的复制操作, 比 arraybyte stream 快
        System.arraycopy(imagebytes, index * IImageEffectsCacheService.
PERPATCH_IMAGE_SIZE, values, 0, valuesLen);
        binaryJedisCluster.set(key[index], values , "NX" , "EX" ,
redisProperties.getKeyExpiredTime());
    }
    .....

```

有的读者指出这里可以在 Redis Cluster Client 部分使用线程池, 同时读取 Redis 中同一张图片的多个分片部分, 并通过 CountdownLatch (Java 中的并发控制信号) 控制所有分片读取完成后再进行合并。但实际上这种读取方式在 Redis Cluster 集群节点较少的时候意义不大。主要还是因为 Redis 服务节点工作在单线程状态下, 完全依靠操作系统的多路复用 I/O 模型、自身实现的事件分离器、全内存态数据存储和内部的数据结构实现来保证吞吐性能。但是随着 Redis Cluster 集群中 Master 节点的增多, 以上所描述的 Client 多线程方式就有性能优势了, 这是因为通过 Jedis Client CRC16 Hash 算法, 同一个图片的不同 byte 段会分配到更多不同的 Redis 节点上进行存储, 这样就能够实现同一张图片不同 byte 段的同时读取了。

## 同类好书



当第一次阅读作者的样稿后，我就决定一定要把这本书推荐给我的技术团队，不管他们使用的是Java还是PHP，是一名初级工程师还是一名资深的专家。我本人从事金融行业高性能、高可靠系统设计多年，非常惊喜地看到，本书由浅入深地讲解了我曾在工作中踩过的多数“坑”。

质数金服CEO 邓柯

作为一名有十多年工作经验的IT从业者，本人一直希望看到这样一本书，能够将宏观理论与微观操作进行结合，为业内人员提供一个在完整复杂业务模型下搭建高性能、高可用性的整体解决方案的书籍，很高兴我今天读到了！

车上码创始人&CEO 应恒

互联网服务系统的构建和优化，既需要扎实的计算机科学理论基础，又需要丰富的工程实践。本书内容大部分来自作者的日常工作，覆盖面广、逻辑性强，是每一个追求卓越的互联网系统架构师案头必备的宝典。

亦非云联合创始人&副总裁 肖友能

作为作者多年的同事、好友，当听说他要出书，且是一本汇集自身多年技术经历的实战经验型书籍时，顿觉甚好！庄子曾说过：“吾生也有涯，而知也无涯”。作为读者，如果可以通过此书加深自己对各种实战技术的理解，改造自己的思维方式，那么便可实现对“无涯”知识事半功倍地获悉。

英博格.EBGbot创始人&CEO 刘克

通读本书之后更有相见恨晚之意。本书理论朴实而实战丰富，对如何构建一个健壮的高性能服务系统这个复杂主题做了全面而又通俗易懂的精彩介绍，可谓一气呵成，酣畅淋漓地打通了一名码农快速成长为软件架构师的任督二脉，让码农的人生平添一份仗剑走天涯的快意。

英博格.EBGbot CSO 张俭

当前，全球已经进入了大数据和人工智能时代，而它们所依赖的基础是大规模高性能的分布式系统。本书作者从十多年的实践出发，分享了构建高性能服务系统的设计理念和实战经验，并引导读者进行场景实战。本书是一本架构高大上而实操接地气不可多得的好书，推荐阅读。

清数科技创始人&《架构大数据》作者 赵勇



博文视点Broadview



@博文视点Broadview



策划编辑：付睿 @Winnie说说  
责任编辑：石倩  
封面设计：侯士卿

上架建议：系统架构、软件架构

ISBN 978-7-121-31509-1



9 787121 315091 >

定价：89.00元